Universidad Carlos III de Madrid
Escuela Politécnica Superior

Paralelización de un Sistema de Diseño de Redes Neuronales Artificiales para la predicción de series temporales

Ingeniería en Informática
Proyecto Fin de Carrera

Autor: Borja Prior González
Tutor: Juan Peralta Donate

Tribunal nombrado por el Mgfco. y Excmo. Sr. Rector de la Universidad Carlos III de Madrid, el día ……. de …….. de 2011.

Presidente: D. ……

Vocal: D. ……

Vocal: D. ……

Vocal: D. ……

Secretario: D. ……

Realizado el acto de defensa y lectura del Proyecto Fin de Carrera el día …….. de …….. 2011 en ……

Calificación: ……

EL PRESIDENTE LOS VOCALES

EL SECRETARIO

Agradecimientos

Quisiera aprovechar la oportunidad que se me brinda para agradecer a todas esas personas que me han apoyado tanto durante este tiempo y que me han dado la oportunidad de desarrollarme, tanto intelectual como personalmente. En primer lugar, y como no podría ser de otra forma, agradecer a toda mi familia, que han sido un pilar tanto en los momentos buenos como en los más difíciles, donde nunca me han fallado.

En segundo lugar, quisiera agradecer a todos aquellos amigos que siempre han confiado en mí, por su apoyo y amistad que siempre te fortalecen en los momentos más difíciles, así como a todos los profesores que me han encaminado hacia este momento durante toda mi vida.

Por último, pero no por ello menos importante, a Juan Peralta, mi tutor de proyecto, por ofrecerme este proyecto que me permite culminar la carrera y poder comenzar a luchar por nuevas metas.

Gracias a todos.

Índice General

Índice de Figuras

Índice de Tablas

Página en blanco intencionadamente

1. Introducción

Una de las preocupaciones inherentes al ser humano es conocer que le deparará el futuro. Es por ello que, con el transcurso del tiempo, la sociedad ha llevado a cabo diferentes maneras de “adivinar” el futuro.

La sociedad se fue dando cuenta poco a poco de ciertos comportamientos regulares existentes en la naturaleza. Estas regularidades resultaban obvias y observables en el movimiento de los cuerpos celestes a través del firmamento. Es por ello, que la astronomía es una de las primeras ciencias conocidas. Se le dotó de una firme base matemática, dada por Newton, hace ya más de 300 años y todavía hoy en día se hace uso de estas teorías, entre ellas la teoría de la gravedad, para poder predecir el movimiento de casi todos los cuerpos celestes.

Al igual que ocurría con los astros, se llegó a la conclusión de que otros fenómenos naturales también obedecen leyes científicas definidas. Esto llevó a la idea del determinismo científico, cuya primera defensa en público fue llevada a cabo por el científico francés Laplace.

Por lo tanto, la teoría de que el estado del universo en un instante dado determina el estado en cualquier otro momento ha sido uno de los principales dogmas de la ciencia desde los tiempos de Laplace. Esta teoría defiende que podemos predecir el futuro, al menos en principio. Pero en cuanto a la práctica se refiere, nuestra posible capacidad de predicción del futuro se encuentra limitada por las ecuaciones que lo rigen y más concretamente por su complejidad, además del hecho de que éstas normalmente presentan una propiedad denominada caos.

Sobre este tema Einstein no compartía la idea de una aparente aleatoriedad en la naturaleza[42]. Su opinión se resumía en su famosa frase “*Dios no juega a los dados*”. Él iba más allá y defendía que la incertidumbre era tan sólo provisional y que existía una realidad subyacente en la que las partículas tendrían posiciones y velocidades bien definidas y se comportarían de acuerdo con leyes deterministas, en consonancia con Laplace. Esta realidad podría ser conocida por Dios, pero la naturaleza cuántica de la luz nos impediría verla, excepto tenuemente a través de un cristal.

Hoy en día los investigadores e ingenieros hacen uso de diferentes técnicas y estudios matemáticos, estadísticos e informáticos para llevar a cabo la compleja tarea de predecir, si no el futuro, al menos parte de él.

Ejemplos de estas técnicas son los métodos estadísticos de Holt-Winters [43] (en los sesenta) o la metodología ARIMA [44] (en los setenta). Más recientemente, diversos métodos de Inteligencia Computacional (IC) han sido aplicados a la predicción de series temporales con diferentes resultados. Entre dichos métodos, cabe destacar los Sistemas Inmunes Artificiales (SIA) [45], Máquinas de Soporte Vectorial (MSV) [46], Redes Neuronales Artificiales

(RNA) [47], Técnicas Difusas [48,49] o incluso sistemas híbridos combinando cualquiera de las anteriormente comentadas con Computación Evolutiva (CE) [50,51].

El proyecto se basa en la predicción de series temporales mediante Redes Neuronales Artificiales (RNA). Las RNA proporcionan una metodología que puede resolver problemas no lineales que son difíciles de resolver por medio de técnicas tradicionales. A menudos las series temporales muestran variabilidad espacial y temporal, y sufren la no linealidad de los procesos físicos. Las RNA son modelos flexibles capaces de extraer la relación entre las entradas y las salidas de un proceso y no requieren un conocimiento “a priori”, son además capaces de llevar a cabo un modelado no lineal y a menudo con tolerancia ante el “ruido”.

Sin embargo, las RNA requieren un alto grado de cómputo. Por lo que, los resultados de éstas en muchas ocasiones se obtienen tras cálculos de muchos días e incluso años.

El proyecto se basa en realizar la paralelización del cómputo que requiere realizar un sistema de RNA de forma que no se ejecute en un único computador, sino que pueda ser ejecutado en diferentes computadores a la vez permitiendo de esta forma obtener resultados en un tiempo mucho menor y por ende, poder ejecutar el sistema con series temporales de tamaños mayores.

1.1 Objetivos

El objetivo del proyecto consiste en la paralelización de un sistema de generación automática de redes neuronales para la predicción de series temporales. Dicho sistema ha obtenido premios de distinta importancia en torneos a nivel internacional NN3 (2007), NN5 (2008) y NNGC (2009) [52], pero debido a la gran necesidad de cómputo de éste, el tiempo empleado para ejecutarse en una única computadora era su principal hándicap para lograr mejores resultados.

Por lo que el proyecto surgió con el fin de darle solución a su principal hándicap, reducir el tiempo de ejecución del algoritmo empleando la paralelización. La paralelización permitirá ejecutar el algoritmo en un mismo momento en distintas máquinas, aprovechando la capacidad de computación de cada una de ellas, reduciendo significativamente el tiempo de cómputo global del problema.

La paralelización del sistema permitirá por lo tanto, poder realizar predicciones mediante las redes neuronales en un menor tiempo y poder abordar series temporales de tamaños mayores, y por lo tanto, poder llegar a predicciones más acertadas.

1.2 Organización de la Memoria

Con el fin de que se pueda entender dar una visión general de la estructura y contenido del presente documento y del contenido del proyecto, se detalla a continuación la estructuración dada al documento:

- Capítulo 1: Introducción. En el presente capítulo, se da una justificación de porqué se ha decidido llevar a cabo el proyecto, presentando el problema al que se afrontaba y de cómo se ha resuelto, así como dar un detalle del contenido del proyecto.

- Capítulo 2: Estado del Arte. Capítulo en el cuál se ha realizado un recorrido breve por las distintas áreas de conocimiento que se ven relacionadas con el proyecto realizado. El Estado del Arte, pretende dar las nociones básicas de conocimiento para poder comprender el proyecto elaborado.

- Capítulo 3: Implementación Paralelización MPI. En el tercer capítulo, se describe detalladamente cómo se ha resuelto la paralelización del algoritmo genético con el estándar MPI (*Message Passing Interface*).

- Capítulo 4: Implementación Paralelización OpenMP. En el cuarto capítulo, se describe con detalle cómo se ha implementado la paralelización del algoritmo esta vez con el estándar OpenMP.

- Capítulo 5: Experimentación. En el quinto capítulo, se presenta las distintas series temporales con las que se ha llevado a cabo la experimentación, así como, se muestran los distintos tiempos obtenidos en la experimentación realizada. Se irá inicialmente un análisis de los resultados obteniedos tanto en tiempo, *speed-up* como eficiencia con el paradigma MPI y OpenMP. Y finalmente, se realizará una comparativa de ambos.

- Capítulo 6: Conclusiones. En el séxto capítulo, una vez realizado el análisis de los resultados, se describirán las distintas conclusiones a las que se haya podido llegar.

- Capítulo 7: Trabajos Futuros. En el séptimo capítulo, se presentan las distintas líneas de trabajo sobre las que se podría dar continuidad al trabajo realizado con el fin de llegar a conclusiones adicionales e incluso poder llegar a mejorar los resultados obtenidos.

- Capítulo 8: Gestión del Proyecto. En el octavo capítulo, se presenta la planificación realizada para llevar a cabo la ejecución del proyecto, así como, la valoración económica del proyecto, teniendo en cuenta todos los conceptos económicos del proyecto.

- Capítulo 9: Apéndice. Noveno capítulo en donde se han incluido los distintos manuales de instalación de MPI y OpenMP, así como, un manual de usuario en donde se detalla cómo debe llevarse a cabo el lanzamiento del proyecto y cómo se configuran los distintos parámetros tenidos en cuenta en éste.

2. Estado del Arte

2.1 Inteligencia Artificial

A continuación se muestran definiciones de Inteligencia Artificial (IA) extraídas de ocho libros de texto. Las que aparecen en la parte superior se refieren a procesos mentales y al razonamiento, mientras que las de la parte inferior aluden a la conducta. Las definiciones de la izquierda miden el éxito en términos de la fidelidad en la forma de actuar de los humanos, mientras que las de la derecha toman como referencia un concepto ideal de inteligencia, que llamaremos **racionalidad**. Un sistema es racional si hace "lo correcto", en función de su conocimiento.

Sistemas que piensan como humanos	**Sistemas que piensan racionalmente**
"El nuevo y excitante esfuerzo de hacer que los computadores piensen… máquinas con mentes, en el más amplio sentido literal" (Haugeland, 1985)[1] "[La automatización de] actividades que vinculamos en procesos de pensamiento humano, actividades como la toma de decisiones, resolución de problemas, aprendizaje, …" (Bellman, 1978)[3]	"El estudio de las facultades mentales mediante el uso de modelos computacionales" (Charniak y McDermott, 1985)[2] "El estudio de los cálculos que hacen posible percibir, razonar y actuar" (Winston, 1992) [4]
Sistemas que actúan como humanos	**Sistemas que actúan racionalmente**
"El arte de desarrollar máquinas con capacidad para realizar funciones que cuando son realizadas por personas requieren de inteligencia" (Kurzweil, 1990) "El estudio de cómo lograr que los computadores realicen tareas que, por el momento, los humanos hacen mejor" (Rich y Knight, 1991)[5]	"La Intelilgencia Computacional es el estudio del diseño de agente inteligentes" (Puole et al.. 1998) "IA…está relacionada con conductas inteligentes en artefactos" (Nilsson, 1998)[6]

Tabla 1 - Definiciones de Inteligencia Artificial

A lo largo de la historia se han seguido los cuatro enfoques mencionados. Como es de esperar, existe un enfrentamiento entre los enfoques centrados en los humanos y los centrados entorno a la racionalidad. El enfoque centrado en el comportamiento humano debe ser una ciencia empírica, que incluya hipótesis y confirmaciones mediante experimentos. El enfoque racional implica una combinación de matemáticas e ingeniería. Cada grupo al mismo tiempo ha ignorado y ha ayudado al otro. A continuación se revisa cada uno de los cuatro enfoques con más detalle.

2.1.1 Comportamiento humano: el enfoque de la Prueba de Turing

La Prueba de Turing, propuesta por Alan Turing (1950), se diseñó para proporcionar una definición operacional y satisfactoria de la inteligencia. En vez de proporcionar una lista larga y quizá controvertida de cualidades necesarias para obtener inteligencia artificialmente, él sugirió una prueba basada en la incapacidad de diferenciar entre entidades inteligentes indiscutibles y seres humanos. El computador supera la prueba si un evaluador humano no es capaz de distinguir si las respuestas, a una serie de preguntas planteadas, son de una persona o no. El computador debería poseer las siguientes capacidades:

- **Procesamiento de lenguaje natural** que le permita comunicarse satisfactoriamente.

- **Representación del conocimiento** para almacenar lo que se conoce o siente.

- **Razonamiento automático** para utilizar la información almacenada para responder a preguntas y extraer nuevas conclusiones.

- **Aprendizaje automático** para adaptarse a nuevas circunstancias y para detectar y extrapolar patrones.

La Prueba de Turing evitó deliberadamente la interacción física directa entre el evaluador y el computador, dado que para medir la inteligencia es innecesario simular físicamente a una persona. Sin embargo, la llamada Prueba Global de Turing incluye una señal de vídeo que permite al evaluador valorar la capacidad de percepción del evaluado, y también le da la oportunidad al evaluador de pasar objetos físicos “a través de una ventanita”. Para superar la Prueba Global de Turing el computador debe estar dotado de:

- **Visión computacional** para percibir objetos
- **Robótica** para manipular y mover objetos

Estas seis disciplinas abarcan la mayor parte de la IA, y Turing merece ser reconocido por diseñar una prueba que se conserva vigente después de 50 años. Los investigadores del campo de la IA han dedicado poco esfuerzo a la evaluación de sus sistemas con la Prueba de Turing, por creer que es más importante el estudio de los principios en los que se basa la inteligencia artificial que duplicar un ejemplar. La búsqueda de un ingenio que “volara artificialmente” tuvo éxito cuando los hermanos Wright, entre otros, dejaron de imitar a los pájaros y comprendieron los principios de la aerodinámica. Los texto de ingeniería aerodinámica no definen el objetivo de su campo como la construcción de “máquinas que vuelen como palomas de forma que puedan incluso confundir a otras palomas”.

2.1.2 Pensar como un humano: el enfoque del modelo cognitivo

Para poder decir que un programa dado piensa como un humano, es necesario contar con un mecanismo para determinar cómo piensan los humanos. Es necesario penetrar en el funcionamiento de las mentes humanas. Hay dos formas de hacerlo: mediante introspección (intentando atrapar nuestros propios pensamientos conforme éstos van apareciendo) y mediante experimentos psicológicos. Una vez se cuente con una teoría lo suficiente precisa sobre cómo trabaja la mente, se podrá expresar esa teoría en la forma de un programa de computador. Si los datos de entrada/salida del programa y los tiempos de reacción son similares a los de un humano, existe una evidencia de que algunos de los mecanismos del programa se pueden comparar con los que utilizan los seres humanos. Por ejemplo, a Allen Newel y Herbert Simon, que desarrollaron el "*Sistema de Resolución General de Problemas*" (SRGP), no les bastó con que su programa resolviera correctamente los problemas propuestos. Lo que les interesaba era seguir la pista de las etapas del proceso de razonamiento y compararlas con las seguidas por los humanos a los que les enfrentó a los mismos problemas. En el campo interdisciplinario de la ciencia cognitiva convergen modelos computacionales de IA y técnicas experimentales de psicología intentando elaborar teorías precisas y verificables sobre el funcionamiento de la mente humana.

2.1.3 Pensamiento racional: el enfoque de las "leyes del pensamiento"

El filósofo griego Aristóteles fue uno de los primeros en intentar codificar la "manera correcta de pensar", es decir, un proceso de razonamiento irrefutable. Sus silogismos son esquemas de estructuras de argumentación mediante las que siempre se llega a conclusiones correctas si se parte de premisas correctas (por ejemplo: "Sócrates es un hombre; todos los hombres son mortales; por lo tanto Sócrates es mortal"). Estas leyes de pensamiento supuestamente gobiernan la manera de operar de la mente; su estudio fue el inicio de un campo llamado lógica.

Estudiosos de la lógica desarrollaron, en el siglo XIX, una notación precisa para definir sentencias sobre todo tipo de elementos del mundo y especificar relaciones entre ellos (compárese esto con la notación aritmética común, que prácticamente sólo sirve para representar afirmaciones acerca de la igualdad y desigualdad entre números). Ya en 1965 existen programas que, en principio, resolvían cualquier problema resoluble descrito en notación lógica. La llamada tradición Logista dentro del campo de la inteligencia artificial trata de construir sistemas inteligentes a partir de estos programas.

Este enfoque presenta dos obstáculos. No es fácil transformar conocimiento informal y expresarlo en términos formales que requieren notación lógica, particularmente cuando el conocimiento que se tiene es inferior al 100 por 100. En segundo lugar, hay una gran diferencia entre poder resolver un problema "en principio" y hacerlo en la práctica. Incluso problema con apenas una docena de datos pueden agotar los recursos computacionales de cualquier

computador a menos que cuente con alguna directiva sobre los plazos de razonamiento que hay que llevar a cabo primero. Aunque los dos obstáculos anteriores están presentes en todo intento de construir sistemas de razonamiento computacional, surgieron por primera vez en la tradición lógica.

2.1.4 Actuar de forma racional: el enfoque del agente racional

Un agente es algo que razona (agente viene del latín *agere*, hacer). Pero de los agentes informáticos se espera que tengan otros atributos que los distingan de los "programas" convencionales, como que estén dotados de controles autónomos, que perciban su entorno, que persistan durante un periodo de tiempo prolongado, que se adapten a los cambios, y que sean capaces de alcanzar objetivos diferentes. Un agente racional es aquel que actúa con la intención de alcanzar el mejor resultado o, cuando hay incertidumbre, el mejor resultado esperado.

En el caso del enfoque de la IA según las "leyes del pensamiento", todo el énfasis se pone en hacer inferencias correctas. La obtención de estas inferencias correctas puede, a veces, formar parte de lo que se considera un agente racional, ya que una manera racional de actuar es llegar a la conclusión lógica de que si una acción dada permite alcanzar un objetivo, hay que llevar a cabo dicha acción. Sin embargo, el efectuar una inferencia correcta no depende siempre de la racionalidad, ya que existen situaciones para las que no hay nada correcto que hacer y en las que hay tomar una decisión. Existen también formas de actuar racionalmente que no implican realizar inferencias. Por ejemplo, el retirar la mano de una estufa caliente es un acto reflejo mucho más eficiente que una respuesta lenta llevada a cabo tras una deliberación cuidadosa.

Todas las habilidades que se necesitan en la Prueba de Turing deben permitir emprender acciones racionales. Por lo tanto, es necesario contar con la capacidad para representar el conocimiento y razonar basándonos en él, porque ello permitirá alcanzar decisiones correctas en una amplia gama de situaciones. Es necesario ser capaz de generar sentencias comprensibles en lenguaje natural, ya que el enunciado de tales oraciones permite a los agentes desenvolverse en una sociedad compleja. El aprendizaje no se lleva a cabo por erudición exclusivamente, sino que profundizar en el conocimiento de cómo funciona el mundo facilita la concepción de estrategias mejores para manejarse en él.

La percepción visual es necesaria no sólo porque ver es divertido, sino porque es necesaria para poder tener una idea mejor de lo que una acción puede llegar a representar, por ejemplo, el ver un delicioso bocadillo contribuirá a que nos acerquemos a él.

Por esta razón, el estudiar la IA desde el enfoque del diseño de un agente racional ofrece al menos dos ventajas. La primera es más general que el enfoque que proporcionan las "leyes del pensamiento", dado que el efectuar inferencias correctas es sólo uno de los mecanismos existentes para garantizar

la racionalidad. La segunda más afín a la forma en la que se ha producido el avance científico que los enfoques basados en la conducta o pensamiento humano, porque la normal de la racionalidad está claramente definida y es de aplicación general. Por el contrario, la conducta humana se adapta bien a un entorno específico, y en parte, es producto de un proceso evolutivo complejo, en gran medida desconocido, que aún está lejos de llevarnos a la perfección.

2.2 Series Temporales

Una serie temporal es una secuencia de datos, observaciones o valores, medidos en determinados momentos del tiempo, ordenados cronológicamente y, normalmente, espaciados entre sí de manera uniforme[7][8].

Las series suelen representarse mediante un gráfico que muestra su evolución con el tiempo. Cuando se representa una serie se suele prestar atención a una serie de características. El estudio descriptivo de series temporales se basa en la idea de descomponer la variación de una serie en varias componentes básicas. Este enfoque no siempre resulta ser el más adecuado, pero es interesante cuando en la serie se observa cierta tendencia o cierta periodicidad. Hay que resaltar que esta descomposición no es en general única.

Este enfoque descriptivo consiste en encontrar componentes que correspondan a una tendencia a largo plazo, un comportamiento estacional y una parte aleatoria.

Las componentes o fuentes de variación que se consideran habitualmente son las siguientes:

1. **Tendencia**: Se puede definir como un cambio a largo plazo que se produce en relación al nivel medio, o el cambio a largo plazo de la media. La tendencia se identifica con un movimiento suave de la serie a largo plazo.

2. **Efecto Estacional**: Muchas series temporales presentan cierta periodicidad o dicho de otro modo, variación de cierto periodo (anual, mensual,…). Por ejemplo, el paro laboral aumenta en general en invierno y disminuye en verano. Estos tipos de efectos son fáciles de entender y se pueden medir explícitamente o incluso se pueden eliminar del conjunto de los datos, *desestacionalizando* la serie temporal.

3. **Componente Aleatoria**: Una vez identificados los componentes anteriores y después de haberlos eliminado, persisten unos valores que son aleatorios. Se pretende estudiar qué tipo de comportamiento aleatorio presentan estos residuos, utilizando algún tipo de modelo probabilístico que los describa.

El análisis de series temporales comprende métodos que ayudan a interpretar este tipo de datos, extrayendo información representativa, tanto referente a los orígenes o relaciones subyacentes como la posibilidad de extrapolar y predecir

su comportamiento futuro. De hecho uno de los usos más habituales de las series temporales es su análisis para predicción y pronóstico. Por ejemplo, de los datos climáticos, de las acciones de bolsa, o de las series pluviométricas. Resulta difícil imaginar una rama de la ciencia en la que no aparezcan datos que puedan ser considerados como series temporales. Son estudiadas en estadística, procesamiento de señales, econometría y muchas otras áreas.

2.3 Aprendizaje Automático

2.3.1 Introducción

No es habitual comenzar a explicar el tipo de aprendizaje que puede realizar una máquina, al que se llamará aquí *aprendizaje automático*, estudiando los procesos de aprendizaje que son observables en la naturaleza aunque existen algunos precedentes (por ejemplo, [9] y [10]). La razón es que si se quiere buscar un marco cognitivo para explicar el fenómeno del aprendizaje, parece razonable referirse a aquellas conductas observables en los animales y que son identificables como aprendizaje que pueden ayudar a explicar de manera más completa un proceso tan complejo como el que nos ocupa.

No basta con sólo intentar explicar qué y cómo se aprende en términos de procesos generales tales como asociaciones, abstracción de prototipos, pruebas de hipótesis, inducción, razonamiento analógico, asimilación, generalización o diferenciación. Hay razones para afirmar que existe cierto sesgo en las estructuras cognitivas que se emplean en cada una de las tareas mencionadas y que dicho sesgo depende directamente de características específicas del dominio donde algo ha de ser aprendido.

Aprendizaje animal

Cuando los organismos se ajustan o adaptan al conjunto de estímulos que provienen del entorno, es decir, reciben información y la almacenan con el fin de reutilizarla en situaciones o patrones de estímulos semejantes, se puede decir que aprenden. En particular nos referimos a los animales ya que, desde un punto de vista del aprendizaje, y a diferencia de las plantas, son móviles y activos. De hecho, se puede afirmar que los animales son máquinas que presentan una conducta predatoria y tienen que moverse para localizar su alimento y conseguirlo o, al menos, están dotados de órganos especializados para ello, como es el caso de muchos seres marinos que no se mueven pero son capaces de filtrar el agua que absorben para alimentarse. Este énfasis en el movimiento de los animales es fundamental para explicar la existencia de diferentes tipos de aprendizaje. Es posible imaginar dos situaciones muy diferentes:

- El animal se mueve de manera aleatoria; en este caso el mismo movimiento debe acercarle al medio en el cual pueda proveerse de todo lo necesario para su subsistencia (agua, oxígeno, comida, etc) y

asegurar su desarrollo y la supervivencia de una proporción significativa de los series de su especie.

- Si un movimiento aleatorio no es suficiente debe existir un movimiento dirigido para lo que debe haber órganos especiales de detección de objetos en el entorno y guiar al animal hacia ellos, de forma directa o indirecta. La mayor precisión de estos movimientos depende de la evolución causada por la presión generada por la competencia con otras especies por unos recursos acotados en un determinado hábitat ecológico.

Una parte significativa de la conducta de muchos animales parece ser fuertemente arraigada e influenciada por la experiencia, de tal forma que puede ser descrita como innata, instintiva o, simplemente, no aprendida. Es decir, muchos animales están de alguna forma programados de manera innata y, cuando perciben alteraciones en su entorno y cambian sus patrones de conducta como resultado de esta percepción – se puede decir que aprenden. Desde este punto de vista, el aprendizaje puede ser definido como la organización (o reorganización) de la propia conducta (ante una situación o un patrón de estímulos) como resultado de una experiencia individual. La definición anterior indica un mínimo de características que un fenómeno debe presentar para poder ser clasificado como un ejemplo de aprendizaje y evitar en la medida de lo posible la confusión que puede causar intentar definirlo a partir de la manera en que el proceso de aprendizaje se ha realizado. Esta definición permanece inmutable aún en caso de necesitar explicar este proceso en situaciones en las que la conducta se ve continuamente modificada por la adquisición de nuevos conocimientos. Algunos autores han sugerido la necesidad de explicar el aprendizaje animal en términos conductistas y después en términos cognitivos, si es posible.

Frecuentemente en la literatura, se considera al aprendizaje como un proceso adaptativo, es decir, que se manifiesta mediante cambios suaves, incrementales. Se considerará aquí la adaptatividad del aprendizaje como la medida de ajuste de una conducta. Además, se asume que lo aprendido permanece en memoria durante períodos relativamente largos, de manera que asegura la observabilidad de la conducta aprendida (como algo relativamente estable).

Lo que un animal puede aprender no depende solamente de su capacidad para ello, ya que existen muchas otras restricciones y limitaciones que moldean esta capacidad. Así, dadas estas limitaciones, un animal está más predispuesto a reaccionar a un tipo de estímulos que a otros, puede aprender más de éstos que de aquellos. Aún más, todos los estímulos a los que un individuo responde en un cierto contexto pueden no ser efectivos para producir una conducta de aprendizaje en otros contextos.

Como consecuencia, el aprendizaje debe considerarse como una de las características más apreciables en un sistema.

Tipos de aprendizaje animal

Resulta casi imposible dividir los diversos tipos de aprendizaje en categorías mutuamente exclusivas, que puedan ser definidas exactamente y cubran todo el espectro del aprendizaje animal. Aquí no se pretende ser exhaustivo y tan sólo se busca dar una visión de conjunto del estado del arte.

- **Habituación**: es un tipo de aprendizaje que consiste en una respuesta que decae ante un conjunto de estímulos repetidos (o continuos) no asociados a ningún tipo de recompensa o esfuerzo. La habituación se puede caracterizar como asociada a un estímulo específico y su relativa permanencia la distingue de manifestaciones temporales como la fatiga o la adaptación sensorial. La habituación implica una tendencia a borrar todo tipo de respuesta ante un estímulo que no tiene importancia para la supervivencia. A pesar de ser el tipo más simple de aprendizaje resulta muy efectivo, especialmente en los organismos más simples, ya que sirve como filtro a conjuntos de estímulos que no son relevantes evitando la innecesaria especialización de algún órgano. No obstante, la habituación como mecanismo de aprendizaje está presente en todos los tipos de organismos, independientemente de su complejidad. Sin embargo, los mecanismos que subyacen en el proceso de habituación se vuelven más elaborados cuando los organismos devienen más complejos.

- **Aprendizaje asociativo**: Frecuentemente en los entornos en los que los animales se mueven, un evento permite predecir, con cierta confianza, la ocurrencia (o no ocurrencia) de otro. La aparición de ciertos rasgos en el paisaje pueden indicar el cambio de estación, el cambio de comportamiento de algunos individuos de especie puede anunciar la temporada de celo, o la ingestión de alguna planta (o animal) puede causar alguna enfermedad (o producir consecuencias benéficas). Un animal que conoce esas relaciones puede sacar provecho anticipándose a esos eventos y así comportarse apropiadamente. Pero, ¿cómo se adquiere ese conocimiento?

  A pesar de que el concepto de asociación – la conexión entre un estímulo y una respuesta que no ha existido antes en la conducta de un organismo – tiene una larga historia que puede ser trazada fácilmente, y a pesar de que en la década de los 1880 ya se aplicaba este concepto en los estudios experimentales sobre el aprendizaje humano, los estudios psicológicos modernos del aprendizaje animal asociativo no comenzaron hasta el final del sigo XIX. En ese momento un grupo de psicólogos rusos comenzó a dar las primeras explicaciones sobre cómo las conductas adquiridas, y, probablemente, también las heredadas, pueden ser modificadas y adaptadas mediante su asociación a un nuevo estímulo durante el proceso de entrenamiento (aprendizaje).

- **Condicionamiento**: Los estudios de I. Pavlov sobre la digestión, usando perros, le convirtieron en el investigador ruso más influyente en el final del siglo pasado y sus experimentos dieron lugar a la formulación de la teoría del reflejo condicionado, o condicionamiento clásico.

Esencialmente, la noción de condicionamiento clásico denota el proceso mediante el cual un animal adquiere la capacidad de responder a un estímulo determinado con la misma acción refleja con que respondería a otro estímulo condicionante (refuerzo o recompensa) cuando ambos estímulos se presentan concurrentemente (o sobrepuestos en una secuencia) un cierto número de veces.

- **Aprendizaje mediante prueba y error**: Este tipo de aprendizaje se identificó al observar la conducta de ciertos animales que obtienen recompensas después de realizar con éxito ciertas tareas. En esas situaciones, los animales permanecen siempre activos y su atención se fija primero aquí y luego allá probando todas las posibilidades imaginables hasta que de manera más o menos accidental resuelve con éxito la tarea y obtiene la recompensa. Esto a pesar de no existir una relación entre las acciones realizadas y la superación de la prueba. El aprendizaje mediante prueba y error requiere entonces la existencia del refuerzo (o recompensa) para animar la selección de la respuesta adecuada de entre una variedad de conductas en una situación determinada, hasta que finalmente se establece una relación entre el estímulo o situación y una respuesta correcta para obtener una recompensa.

En este caso el refuerzo está precedido por el estímulo y la respuesta requerida, lo que no ocurre forzosamente en el condicionamiento clásico. A este tipo de aprendizaje se le ha dado muchos otros nombres, tales como condicionamiento operante, condicionamiento instrumental, etc.

- **Aprendizaje latente**: El aprendizaje latente es un tipo de aprendizaje asociativo que tiene lugar en ausencia de recompensa. Un experimento clásico es el realizado con ratas que son dejadas en libertad en un laberinto durante varios días sin ningún tipo de recompensa. Cuando su aprendizaje es comparado con otro grupo que no ha estado en el laberinto y comienza a ser recompensado inmediatamente, los resultados del primer grupo son sorprendentes: aprenden más rápidamente y con menos errores que el segundo grupo. De aquí se desprende que el primer grupo aprendió algo durante su estancia en el laberinto que permanece latente hasta que es necesitado.

- **Imitación**: La imitación ha sido frecuentemente considerada como una evidencia de la existencia de conductas altamente reflexivas, a pesar de que diversos fenómenos son incluídos bajo la etiqueta de imitación.

Uno de los tipos de imitación más comunes es la denominada facilitación social (*social facilitation*) que describe un patrón de conducta ya existente en el repertorio de un individuo, ya que éste realiza cuando la misma conducta es realizada por otros miembros de su especie. Por ejemplo, en los humanos, bostezar.

Pero la verdadera imitación, que implica copiar una conducta, acción o expresión nueva o que resulta imposible de aprender si no es copiada de otro individuo, se presenta especialmente en los humanos y en algunos

chimpancés y monos. En particular, uno puede imaginar un ejemplo de este tipo de aprendizaje el que ocurre cuando un individuo es entrenado para realizar un salto de pértiga. En otros animales, como los felinos y otros cazadores, el aprendizaje de cómo matar certeramente una presa es realizado mediante la imitación de los padres y reforzado mediante los juegos.

Si bien, tal como se ha definido, la imitación significa una copia consciente de una conducta, acción o expresión realizada por otro individuo, también está asociada a un intento de obtener un provecho de la experiencia de otro.

El aprendizaje aplica inferencias a determinada información para construir una representación apropiada de algún aspecto relevante de la realidad o de algún proceso.

Una metáfora habitual en el área del aprendizaje automático – dentro de la Inteligencia Artificial – es considerar la resolución de problemas como un tipo de aprendizaje que consiste – una vez resuelto un tipo de problema – en ser capaz de reconocer la situación problemática y reaccionar usando la estrategia aprendida. Actualmente la mayor distinción que se puede trazar entre un animal y un mecanismo de resolución de problemas es que ciertos animales son capaces de mejorar su actuación, en un amplio conjunto de tareas, como resultado de haber solucionado un cierto problema.

Se asume, en este enfoque, que un agente autónomo debe tener la capacidad de realizar una misma tarea de varias maneras, si es posible, y dependiendo de las circunstancias. Debe ser capaz de tomar decisiones sobre cuál es el curso más apropiado que debe seguir la resolución de un problema y modificar estas decisiones cuando las condiciones así lo requieran. Por esto, uno de los objetivos centrales de esta área es construir sistemas (agentes) que sean capaces de adaptarse – dinámicamente y sin un entrenamiento previo – a situaciones nuevas y aprender como resultado de resolver el problema (o problemas) que estas situaciones presentan.

El aprendizaje automático, también llamado aprendizaje artificial [11], es un área de interés muy desarrollada en la IA. En otras áreas afines como la biología, la psicología y la filosofía también se ha investigado la naturaleza de la habilidad de aprender referida a sistemas biológicos y al hombre en particular.

Como ya se ha visto, aprendizaje es un término muy general que denota la forma, o formas, en la cual un animal (o una máquina) aumenta su conocimiento y mejora sus capacidades de actuación (*performance*) en un entorno. De esta manera, el proceso de aprendizaje puede ser visto como un generador de cambios en el sistema que aprende – que por otra parte ocurren lentamente, adaptativamente - y que pueden ser revocados o ampliados. Estos cambios se refieren no sólo a la mejora de las capacidades y habilidades para realizar tareas sino que también implican modificaciones en la representación de hechos conocidos.

En este contexto, se dice que un sistema que aprende de forma automatizada (o aprendiz) es un artefacto (o un conjunto de algoritmos) que, para resolver problemas, toma decisiones basadas en la experiencia acumulada – en los casos resueltos anteriormente – para mejorar su actuación. Estos sistemas deben ser capaces de trabajar con un rango muy amplio de tipos de datos de entrada, que pueden incluir datos incompletos, inciertos, ruido, inconsistencias, etc.

Nuestra primera caracterización del proceso del aprendizaje automático es:

Aprendizaje = *Selección* + *Adaptación* [40]

Visto así, el aprendizaje automático es un proceso que tiene lugar en dos fases. Una en la que el sistema elige (selecciona) las características más relevantes de un objeto (o un evento), las compara con otras conocidas – si existen – a través de algún proceso de cotejamiento (*pattern matching*) y, cuando las diferencias son significativas, adapta su modelo de aquel objeto (o evento) según el resultado del cotejamiento. La importancia del aprendizaje, como se ha dicho, reside en que sus resultado habitualmente se traducen en mejoras en la calidad de actuación del sistema. Un sistema artificial que aprende puede emplear técnicas muy diversas para aprovechar la capacidad de cómputo de un ordenador, sin importar su relación con los procesos cognitivos humanos. Estas técnicas incluyen métodos matemáticos muy sofisticados, métodos de búsqueda den grandes bases de datos, etc. que requieren la creación (o modificación) de estructuras de representación del conocimiento adecuadas para agilizar la identificación de los hechos relevantes.

Una de las motivaciones más importantes en el diseño y construcción de sistemas de aprendizaje automático residen en el hecho de que en muchos dominios la experiencia es escasa, y la codificación del conocimiento que la describe es limitada, fragmentaria y, por lo tanto, incompleta o casi inexistente. Además, dotar a un agente de todo el conocimiento necesario es una tarea muy compleja, costosa, que toma mucho tiempo y en la cual la eliminación de los posibles errores introducidos es difícil y requiere una atención especializada. En el caso de los humanos son necesario cinco o seis años para aprender las habilidades motoras básicas y los rudimiento del lenguaje, y entre 12 a 20 años para manipular conceptos complejos, aprender un oficio, las convenciones culturales e históricas, etc. Además, el aprendizaje en los humanos es personalizado.

2.3.2 Paradigmas del aprendizaje automático

Según el tipo de selección y adaptación (transformación) que un sistema realiza sobre la información disponible es posible identificar varios paradigmas del aprendizaje automático. Esta clasificación ha evolucionado rápidamente en la última década.

- **Aprendizaje deductivo**

  Este tipo de aprendizaje se realiza mediante una secuencia de inferencias deductivas usando hechos o reglas conocidas. A partir de los hechos conocidos nuevos hechos o nuevas relaciones son lógicamente derivadas. En este tipo de sistemas la monotonicidad de la teoría definida por la base de conocimientos es importante.

- **Aprendizaje analítico**

  Los métodos usados en este tipo de aprendizaje intentan formular generalizaciones después de analizar algunas instancias en términos del conocimiento del sistema. En contraste con las técnicas empíricas del aprendizaje – que normalmente son métodos basados en las similitudes – el aprendizaje analítico requiere que se proporcione al sistema un amplio conocimiento del dominio. Este conocimiento es usado para guiar las cadenas deductivas que se utilizan para resolver nuevos problemas. Por tanto, estos métodos se centran en mejorar la eficiencia del sistema, y no en obtener nuevas descripciones de conceptos, como se hace del aprendizaje inductivo.

- **Aprendizaje analógico**

  Este tipo de aprendizaje, intenta emular algunas de las capacidades humanas más sorprendentes: poder entender una situación por su parecido con situaciones anteriores conocidas, poder crear y entender metáforas o resolver un problema notando su posible semejanza con otros vistos anteriormente adaptando (transformando) de forma conveniente la solución que se encontró para esos problemas. Este tipo de sistemas requiere una gran cantidad de conocimiento. Algunos autores consideran que el aprendizaje analógico es una especialización del aprendizaje por explicación.

- **Aprendizaje inductivo**

  Es el paradigma más estudiado dentro del aprendizaje automático. Normalmente, estos sistemas carecen de una teoría del dominio, es decir, no conocen a priori los objetos con los que tratan a su cantidad. Trata problemas como inducir la descripción de un concepto a partir de una serie de ejemplos y contraejemplos del mismo, o determinar una descripción jerárquica o clasificación de un grupo de objetos.

- **Aprendizaje mediante descubrimiento**

  El tipo de Descubrimiento es una forma restringida de aprendizaje en la cual un agente adquiere conocimientos sin la ayuda de un profesor. Este proceso ocurre cuando no existe ninguna “fuente” disponible que posea el conocimiento que el agente busca. Un tipo particular de Descubrimiento se lleva a cabo cuando un agente intenta agrupar objetos que supone del mismo subconjunto.

- **Algoritmos genéticos**

  Los algoritmos genéticos están inspirados en las mutaciones y otros cambios que ocurren en los organismos durante la reproducción biológica de una generación a la siguiente y el proceso de selección natural de Darwin. El problema principal que trata de resolver es el descubrimiento de reglas y la dificultad mayor con que se encuentra es la asignación de crédito a las mismas. Este último punto consiste en valorar positiva o negativamente las reglas según lo útiles que sean al sistema.

- **Conexionismo**

  Otra manera de concebir un sistema de aprendizaje automático es el denominado enfoque conexionista. En esta aproximación el sistema es una red de nodos interconectados, que tiene asociada una regla de propagación de valores, y cuyos arcos están etiquetados con pesos. Ante un conjunto de ejemplos el sistema reacciona modificando los pesos de los arcos. Se dice que el sistema aprende si adapta los pesos de las conexiones de tal manera que le llevan a dar la salida correcta ante todas (o la mayoría) de las entradas que se ofrezcan.

Otra posible clasificación de los métodos de aprendizaje explorados en IA, considerando el tipo de estrategia y las ayudas que recibe un sistema de aprendizaje, es:

- **Supervisados**

  La suposición fundamental de este tipo de método es que los ejemplos proporcionados como entradas son necesarios para cumplir las metas del aprendizaje. Es como aprender con un profesor. En este tipo de método se dan ejemplos y se clasifica de qué concepto lo son.

- **No Supervisados**

  Son diseñados para desarrollar nuevos conocimientos mediante el descubrimiento de regularidades en los datos (*data-driven*). Estos métodos son dirigidos por las metas (*goal-driven*).

- **Mediante refuerzos**

  Este método de aprendizaje está a medio camino entre los dos anteriores. Al sistema se le proponen problemas que debe solucionar. El aprendizaje se realiza únicamente con una señal de refuerzo proporcionada por un profesor o por el entorno como indicador de si se ha resuelto correctamente el problema.

2.4 Algoritmos Genéticos

Los principios básicos de los algoritmos genéticos fueron establecidos por J.H. Holland en [12]. En dicho libro se pretendía describir un nuevo paradigma de computación basado en los principios establecidos por Darwin en "*El origen de las especies*". Dicho paradigma no sólo se aplicaba a campos como la optimización de funciones y la búsqueda (principales aplicaciones de los primeros algoritmos genéticos), sino que sus aplicaciones abarcaban aspectos como modelos económicos, la teoría de juegos, la de modelización de los sistemas nervioso e inmunitario, etc. Posteriormente, han aparecido numerosos trabajos sobre algoritmos genéticos, principalmente en los aspectos referentes a la optimización y a la búsqueda. Actualmente constituyen un campo de trabajo muy activo, tanto en su estudio teórico como en sus diversas aplicaciones, que en el caso de la Inteligencia Artificial hacen de puente entre los sistemas clásicos (simbólicos) y los no clásicos.

Los algoritmos genéticos son procedimientos de búsqueda adaptativos basados en principios derivados de la evolución en poblaciones naturales. Se caracterizan por:

- Una **población** de estructuras (**individuos**) que representan soluciones candidatas del problema a resolver.
- Un mecanismo de selección **competitiva** basada en la bondad (relativa a la población) de cada individuo.
- **Operadores genéticos** idealizadas que modifican las estructuras seleccionadas para crear nuevos individuos.

```
Algoritmo Algen:
t := 0
inicializa I(t)
evalúa I(t)
mientras finalización hacer
selecciona I`(t) de I(t)
modifica I`(t) para obtener I(t+1)
evalúa I(t+1)
t:=t+1
fmientras
falgoritmo
```

Figura 1 – Algoritmo genético canónico

Todo ello permite al algoritmo genético explotar el conocimiento acumulado durante la búsqueda de tal manera que se consiga un equilibrio entre la necesidad de explorar nuevas partes del espacio de búsqueda y la necesidad de centrarse en zonas de alta adaptación del mismo.

2.4.1 El algoritmo genético canónico

En la presente sección se presenta un esquema general de algoritmo genético (Ilustación 0.1) y se describen brevemente sus elementos principales.

El algoritmo genético actúa sobre una población de cadenas de longitud fija[1] *L* sobre un alfabeto ∑ que representan soluciones candidatas del problema a resolver (denominadas *cromosomas*). De cada una de estas cadenas o individuos se dispone de una medida de su **nivel de adaptación** (que, de alguna manera, mide su grado de aproximación a la solución real del problema y que en ocasiones se le denomina *fitness*) denotado por *f*, es decir:

$$f : \sum^{L} - \rightarrow R^{+} \cup ()$$

A esta función se la conoce como **función de adaptación** o **de adecuación** (*fitness function*).

Selección

Esta primera fase del ciclo del algoritmo genético tiene por finalidad conseguir que sean los mejores individuos de la población los que se reproduzcan. Para ello, se utiliza una política que prima la selección de los individuos con mejor función de adaptación, es decir, la probabilidad de entrar en el proceso de reproducción es directamente proporcional a la función de adaptación (posiblemente transformada). Cabe destacar que no son los mejores individuos los que pasas de una generación a otra sino que lo que sobrevive de una a otra es su material genético ya que, en general, no se copian directamente sino que, mediante la aplicación de los diversos operadores genéticos, se obtienen descendientes que formarán la población en la siguiente generación.

Modificación – Operadores genéticos

Sobre los individuos seleccionados se aplican los llamados operadores genéticos. Su función es generar nuevos puntos del espacio de búsqueda. Como éstos han sido obtenidos a partir de información procedente de los mejores individuos de la generación actual, es bastante probable que pertenezca a zonas con altos niveles de adaptación ya que poseen algunas características comunes a ellos.

Entre los operadores genéticos más usuales cabe destacar:

Combinación (*crossover*): Combina la información procedente de dos individuos para generar otros dos.

Mutación (*mutation*): Dado un individuo, modifica aleatoriamente una de las posiciones de su cadena. Usualmente se aplica, después del operador anterior, a cada uno de los individuos generados.

Ambos operadores se aplican probabilísticamente sobre el conjunto de individuos seleccionados. La población se completa mediante la copia directa de los individuos que, aún habiendo sido seleccionados, no han intervenido en la generación de nuevos puntos.

El operador de combinación

El operador de combinación (*crossover*) es el principal responsable del funcionamiento de un algoritmo genético. Su misión consiste en combinar las representaciones de dos individuos dados de la población generando otros dos. Su poder se basa en el hecho de que los nuevos puntos generados tienen alta probabilidad de pertenecer a zonas del espacio de búsqueda en los que el nivel de adaptación es superior a la media. Este hecho se fundamenta en:

1. Los individuos que se combinan han sido previamente seleccionados de acuerdo a su nivel de adaptación.

2. Los individuos generados lo han sido mediante la combinación de las representaciones de los individuos originales.

Se han propuesto diversos operadores de combinación. En la presente sección estudiaremos el más simple (e históricamente el primero de ellos), conocido como **combinación unipuntual** (*1-point crossover*), y dejaremos para una sección posterior el estudio de algunas de las variantes propuestas.

Dados dos individuos de la población (representados por una cadena de longitud *L* sobre el alfabeto ∑), se selecciona al azar (con distribución uniforme) un valor entero *k* entre 1 y *L-1*. Los dos nuevos individuos se crean intercambiando los valores entre las posiciones *k+1* y *L*.

Por ejemplo, sean I_1 e I_2 los individuos:

$$I_1 = 010001 \,||\, 0101$$
$$I_2 = 110101 \,||\, 1100$$

Y la posición seleccionada, entre 1 y 9, es la 6 (indicada por el signo ||), el resultado de la combinación es:

$$I_1^{"} = 0100011100$$
$$I_2^{"} = 1101010101$$

Como puede observarse cada uno de los individuos resultantes comparte características con cada uno de sus ancestros. Por ello, a lo largo de las sucesivas generaciones, aquellas características poseídas por individuos con altos niveles de adaptación tienden a permanecer y, recíprocamente, aquellas pertenecientes a individuos peor adaptados a desaparecer.

El operador mutación

El papel que tiene la mutación en los procesos de evolución, tanto natural como artificial, es usualmente malinterpretado. De hecho se trata de un operador completamente secundario y su única tarea consiste en mantener la diversidad poblacional, de forma que no existan puntos del espacio de búsqueda que sean inalcanzables desde el estado actual. Por ejemplo, supongamos que nuestra población consiste únicamente de las cadenas:

$$I_1 = 0110001011$$
$$I_2 = 1010110001$$
$$I_3 = 0011010011$$

A partir de operaciones de combinación jamás podremos alcanzar individuos que contengan un 0 en la tercera o última posiciones, o un 1 en la antepenúltima, ya que ningún individuo de la población contiene dichos valores en las posiciones mencionadas.

El operador de **mutación**, dado un individuo, selecciona al azar (con distribución uniforme) una posición dentro de la cadena y cambio el valor que contiene. Así queda asegurado que todos los puntos del espacio de búsqueda son potencialmente alcanzables.

Política de sustitución

Existen dos grandes tendencias entre los diversos algoritmos propuestos dependiendo de si, a cada generación, se modifica toda la población o sólo una parte. Estas tendencias conducen a considerar dos grandes familias:

Modelo poblacional (*poblational*): En este caso se seleccionan los individuos a los que se aplicarán los operadores genéticos (según sus niveles de adaptación) y, con probabilidades P_c y P_m, se les aplica combinación o mutación. En el caso del operador de cruzamiento convive nunca con sus ancestros. De hecho es el modelo que se ha descrito hasta el momento.

Modelo de estado fijo (*steady-state*): De entre toda la población se eligen dos individuos para combinar. Posteriormente se elige otro para desaparecer (con probabilidad inversamente proporcional a su nivel de adaptación) y se sustituye por uno de los elementos generados, usualmente el mejor (por ello se dice que este modelo sigue una *política elitista*). Este mecanismo provoca que haya convivencia entre un individuo y sus ancestros.

2.5 Algoritmos Genéticos Paralelos

Los algoritmos genéticos paralelos (o, para abreviar a partir de ahora, AGP) surgen ante la necesidad de cómputo requerida por problemas de extrema complejidad, cuyo tiempo de ejecución utilizando los tradicionales algoritmos genéticos secuenciales es prohibitivo. Es por eso que se buscó la manera de

poder adaptar este tipo de heurísticas a distintas configuraciones de cómputo paralelo:

- Los algoritmos maestro-esclavo.
- Algoritmos de Grano Fino.
- Algoritmos de Grano Grueso.

Clasificación de los Algoritmos Genéticos Paralelos (AGP)

Antes de iniciar la paralelización de un algoritmo, de cualquier tipo, es imprescindible realizar una primera fase de estudio de qué elementos de los que forman el algoritmo son susceptibles de paralelizarse [14]. En el caso de los algoritmos genéticos es claro que la evaluación de la adecuación de los individuos es una tarea cuya paralelización no afecta al comportamiento del algoritmo. Sin embargo, el operador de selección sí debe ser aplicado de forma global a toda la población si queremos que el comportamiento del algoritmo siga siendo el mismo que el de la versión secuencial. Esta primera aproximación para la paralelización de los algoritmos genéticos dará lugar a un conjunto de algoritmos que recibirán el nombre de **algoritmos maestro-esclavo**.

La otra gran aproximación a la paralelización de algoritmos genéticos consiste en dividir la población inicial en subpoblaciones de mayor o menor tamaño que se comuniquen de alguna manera. Este tipo de algoritmos se dice que da lugar a *especies o nichos* de individuos separados. Dentro de esta segunda aproximación podemos distinguir entre algoritmos de grano grueso y algoritmos de grano fino. La diferencia entre ambos es el tamaño de las poblaciones (mayor en los primeros) y la forma en que los individuos interactúan los unos con los otros.

A continuación, se analiza con mayor profundidad cada una de estas aproximaciones a la paralelización de los algoritmos genéticos.

Algoritmos maestro-esclavo

Estos algoritmos trabajan con una única población de individuos que será gestionada por el nodo maestro. La evaluación de la adecuación de los individuos y/o la aplicación de los operadores genéticos puede ser realizada por los nodos esclavos. A cada nodo esclavo le corresponderá una parte de la población total, sobre la cual realizará las operaciones antes citadas. Una vez terminado este proceso, devolverán el resultado al nodo maestro, que realizará la selección de individuos. En la figura 2 se puede encontrar un pequeño esquema de la arquitectura de este tipo de algoritmos.

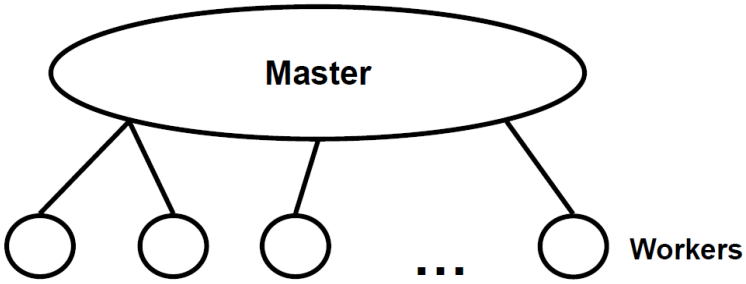

Figura 2 - Representación esquemática de un AGP con arquitectura maestro-esclavo [2]

Normalmente, la operación que se suele implementar en paralelo es la evaluación de la adecuación de los individuos, porque suele ser la más compleja. Además, este valor es independiente del resto de la población, por lo que su implementación es muy sencilla.

El intercambio de información entre los nodos es sencillo: el nodo maestro envía el subconjunto de individuos que corresponde a cada nodo, y éstos le devuelven los valores de adecuación para cada uno de ellos. La comunicación puede ser implementada de dos maneras: síncrona o asíncrona. En la primera de ellas, el nodo maestro espera a recibir los valores de adecuación de todos los individuos para generar la siguiente generación. En la segunda, el algoritmo no espera a que los nodos más lentos envíes sus valores de *fitness*. De este modo conseguimos agilizar el proceso, pero el comportamiento no es exactamente el mismo que el de un algoritmo genético secuencial. La implementación síncrona, en cambio, si mantiene este comportamiento.

Este tipo de algoritmos no especifican nada acerca de la arquitectura subyacente [15], por lo que pueden ser implementados sin problemas en computadores tanto de memoria compartida como de memoria distribuida. En el caso de computadores de memoria compartida, la población podría estar almacenada en memoria, y cada procesador esclavo podría leer directamente los individuos que le hubieran sido asignados. En caso de usar un computador de memoria distribuida, sería el nodo maestro el encargado de asignar y distribuir individuos entre los nodos esclavos y recoger, una vez evaluada, la adecuación de cada uno de ellos.

Algoritmos de grano fino

Este tipo de algoritmos han sido diseñados para ser implementados usando computadores masivamente paralelos. Ahora, la población se encuentra dividida espacialmente entre los distintos procesadores e, idealmente, cada procesador debería albergar un único individuo. El cruce y la selección de individuos se harán entre individuos que pertenezcan al mismo vecindario, formado por un conjunto de individuos adyacentes según la representación antes citada. Sin embargo, se permite el solapamiento entre vecindarios para

propiciar la interacción, aunque leve, entre todos los individuos de la población. La figura 3 recoge un esquema de esta aproximación.

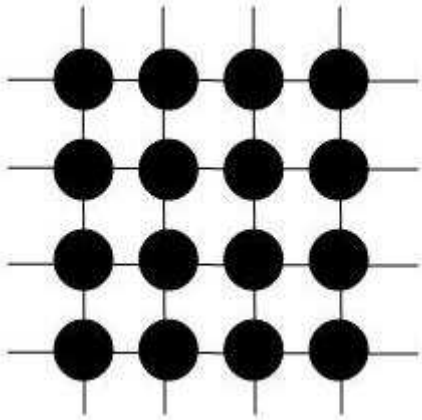

Figura 3 - Representación esquemática de un AGP de grano fino [2]

Algoritmos de grano grueso

Las características más importantes de estos algoritmos son el uso de múltiples poblaciones y la mitigación de individuos entre ellas. Dado que cada una de las poblaciones evolucionan independientemente, el ratio de migración será muy importante de cara a obtener resultados satisfactorios. Se puede observar una posible solución usando este tipo de algoritmos en la figura 4.

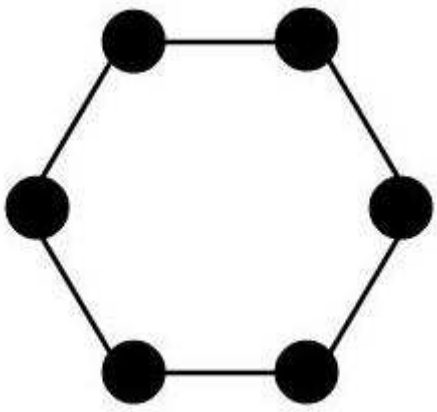

Figura 4 - Representación esquemática de un AGP de grano-grueso [2]

Influencia de las migraciones

Hay que tener en cuenta varios factores en la migración de individuos entre poblaciones. El primero de ellos es la frecuencia con la que éstas se llevarán a cabo. La mayor parte de las implementaciones existentes programan las migraciones a intervalos fijos, o dicho de otra forma, de una forma síncrona. Por otro lado, se puede introducir asincronía en la política de migraciones haciendo que éstas sean efectuadas sólo cuando se produzca un evento.

Otro aspecto muy a tener en cuenta es qué individuo se envía en las migraciones. Existen distintas alternativas a considerar, como el enviar el mejor de cada población a todos sus adyacentes o enviar el mejor individuo a un nodo maestro, que se encargaría de elegir a los mejores individuos entre todos los recibidos y de reenviarlos de nuevo a todas las poblaciones.

Influencia de la topología de comunicación

Aunque muchas veces no se tenga demasiado en cuenta, éste es un factor muy importante a la hora de implementar un algoritmo genético paralelo, ya que determina la velocidad con la que las buenas soluciones encontradas se propagan hacia el resto de poblaciones. Si el grado de conectividad es alto o el diámetro de la red es pequeño (o ambas cosas) las soluciones buenas se expandirán rápidamente a través de todas las poblaciones, favoreciendo así la evolución hacia el óptimo. En caso contrario, las poblaciones estarán más aisladas las unas de las otras, por lo que evolucionarán mucho más despacio, y llevará más tiempo la incorporación de los genes potencialmente buenos a la nueva solución.

Por otro lado, es necesario tener en cuenta que la densidad de conexiones puede influir negativamente en el algoritmo, ya que se supone una sobrecarga importante en el tráfico de la red. Por eso, es muy importante elegir una buena topología para obtener el máximo rendimiento del algoritmo.

Por lo general, se suelen utilizar topologías estáticas, es decir, topologías predefinidas que no cambian según avanza la ejecución del algoritmo.

2.6 Redes de Neuronas Artificiales

El cerebro humano es el sistema de cálculo más complejo que conoce el hombre. El ordenador y el hombre realizan bien diferentes clases de tareas; así la operación de reconocer el rostro de una persona resulta una tarea relativamente sencilla para el hombre y difícil para el ordenador, mientras que la contabilidad de una empresa es tarea costosa para un experto contable y una sencilla rutina para un ordenador básico.

La capacidad del cerebro humano de pensar, recordar y resolver problemas ha inspirado a muchos científicos intentar o procurar modelar en el ordenador el funcionamiento del cerebro humano.

Los profesionales de diferentes campos como la ingeniería, filosofía, fisiología y psicología han unido sus esfuerzos debido al potencial que ofrece esta tecnología y están encontrando diferentes aplicaciones en sus respectivas profesiones.

Un grupo de investigadores ha perseguido la creación de un modelo en el ordenador que iguale o adopte las distintas funciones básicas del cerebro. El resultado ha sido una tecnología llamada Computación Neuronal o también Redes Neuronales Artificiales.

El resurgimiento del interés de esta nueva forma de realizar los cálculos tras dos décadas de olvido se debe al extraordinario avance y éxito tanto en el aspecto teórico como de aplicación que se está obteniendo estos últimos años.

Las Redes Neuronales Artificiales, ANN (*Artificial Neural Networks*) están inspiradas en las redes neuronales biológicas del cerebro humano. Están constituidas por elementos que se comportan de forma similar a la neurona biológica en sus funciones más comunes. Estos elementos están organizados de una forma parecida a la que presenta el cerebro humano.

Las ANN al margen de “parecerse” al cerebro presentan una serie de características propias del cerebro. Por ejemplo las ANN aprenden de la experiencia, generalizan de ejemplos previos a ejemplos nuevos y abstraen las características principales de una serie de datos.

Aprender: adquirir el conocimiento de una cosa por medio del estudio, ejercicio o experiencia. Las ANN pueden cambiar su comportamiento en función del entorno. Se les muestra un conjunto de entradas y ellas mismas se ajustan para producir unas salidas consistentes.

Generalizar: extender o ampliar una cosa. Las ANN generalizan automáticamente debido a su propia estructura y naturaleza. Estas redes pueden ofrecer, dentro de un margen, respuestas correctas a entradas que presentan pequeñas variaciones debido a los efectos de ruido o distorsión.

Abstraer: aislar mentalmente o considerar por separado las cualidades de un objeto. Algunas ANN son capaces de abstraer la esencia de un conjunto de entradas que aparentemente no presentan aspectos comunes o relativos.

2.6.1 Estructura básica de una red neuronal

Analogía con el cerebro

La neurona es la unidad fundamental del sistema nervioso y en particular del cerebro. Cada neurona es una simple unidad procesadora que recibe y combina señales desde y hacia otras neuronas. Si la combinación de entradas es suficientemente fuerte la salida de la neurona se activa. La figura 5 muestra las partes que constituyen una neurona.

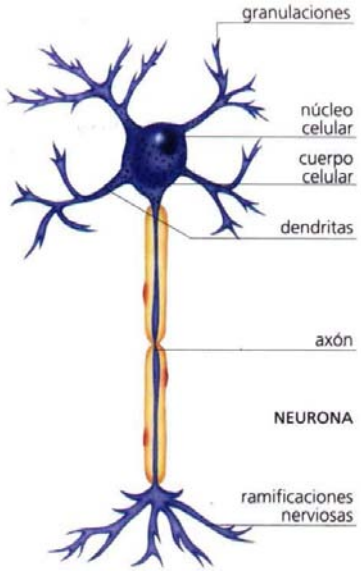

Figura 5 - Componentes de una Neurona

El cerebro consiste en uno o varios billetes de neuronas densamente interconectadas. El axón (salida) de la neurona se ramifica y está conectada a las dendritas (entradas) de otras neuronas a través de uniones llamadas sinapsis. La eficacia de la sinapsis es modificable durante el proceso de aprendizaje de la red.

Redes Neuronales Artificiales

En las Redes Neuronales Artificiales, ANN, la unidad análoga a la neurona biológica es el elemento procesador, PE (*process element*). Un elemento procesador tiene varias entradas y las combina, normalmente con una suma básica. La suma de las entradas es modificada por una función de transferencia y el valor de la salida de esta función de transferencia se pasa directamente a la salida del elemento procesador.

La salida del PE se puede conectar a las entradas de otras neuronas artificiales (PE) mediante conexiones ponderadas correspondientes a la eficacia de las sinapsis de las conexiones neuronales.

La figura 6 representa un elemento procesador de una red neuronal artificial implementada en un ordenador.

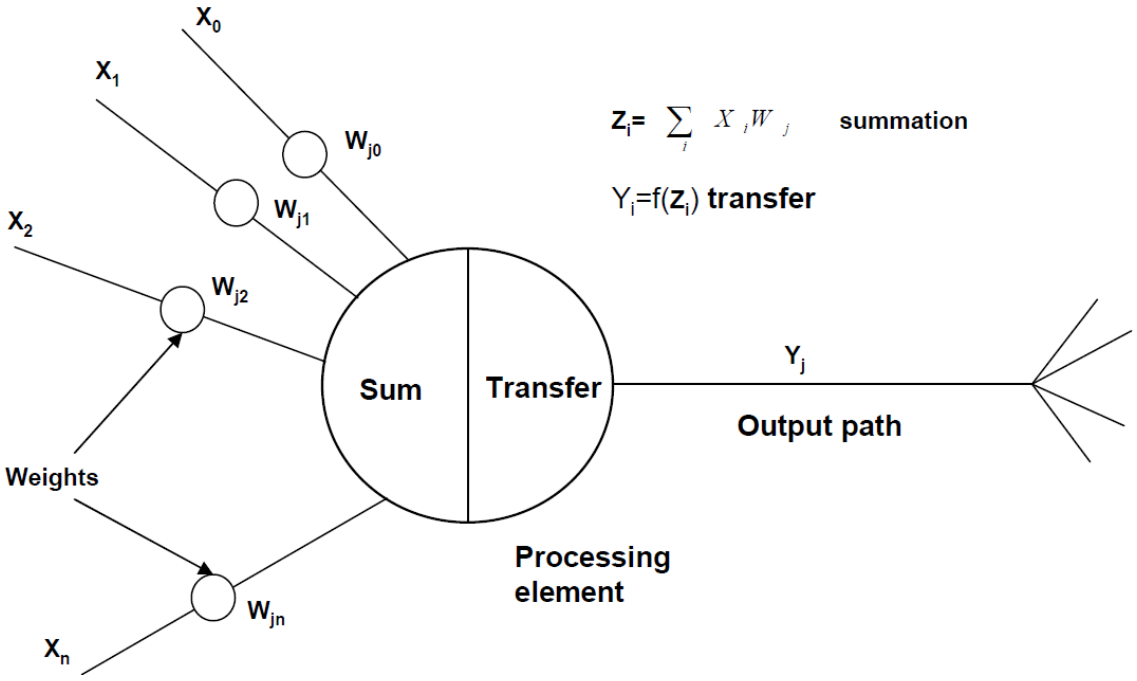

Figura 6 - Diagrama de una Neurona Artificial (PE) [16]

Una red neuronal consiste en un conjunto de unidades elementales PE conectadas de una forma concreta. El interés de las ANN no reside solamente en el modelo del elemento PE sino en las formas en que se conectan estos elementos procesadores. Generalmente los elementos PE están organizados en grupos llamados niveles o capas. Una red típica consiste en una secuencia de capas con conexiones entre capas adyacentes consecutivas[17].

Existen dos capas con conexiones con el mundo exterior. Una capa de entrada, buffer de entrada, donde se presentan los datos a la red, y una capa buffer de salida que mantiene la respuesta de la red a una entrada. El resto de las capas reciben el nombre de capas ocultas. La figura 7 muestra el aspecto de una Red Artificial.

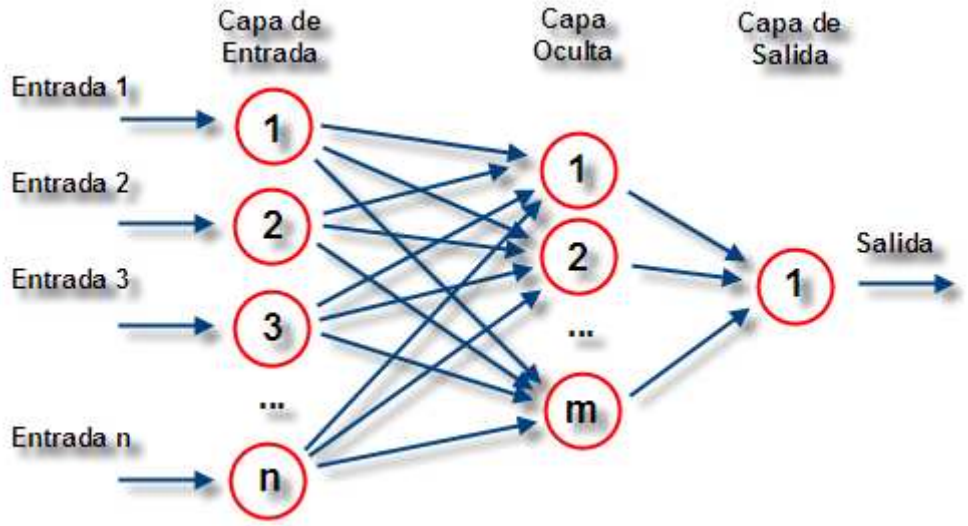

Figura 7 - RNA perceptrón simple con n neuronas de entrada, m neuronas en su capa oculta y una neurona de salida

2.6.2 Computación tradicional y computación neuronal

Programación/Entrenamiento

Las técnicas tradicionales de programación utilizadas para la solución de un problema requieren la creación de un algoritmo. Un algoritmo consiste en una secuencia de instrucciones que indica el modo en el que se debe proceder el sistema basado en un ordenador para lograr el fin perseguido que es la resolución del problema.

El diseño de una secuencia de instrucciones para resolver un problema de contabilidad es relativamente sencillo, mientras que existen muchos problemas del mundo real en los que resulta difícil realizar un algoritmo que resuelva dichos problemas. Por ejemplo, imaginemos desarrollar un programa para cualquiera de los problemas de reconocimiento de imágenes como el rostro de una persona. Hay muchas variantes de la imagen de una persona, como que presente un rostro serio o un rostro alegre, variaciones en general que deben tenerse en cuenta a la hora de diseñar el algoritmo.

Las ANN, a diferencia de los algoritmos que son instrucciones previamente programadas, deben ser previamente entrenadas. Esto significa que a la red se le muestra en su capa de entrada unos ejemplos y ella misma se ajusta en función de alguna regla de aprendizaje.

Arquitectura

Las ANN presentan una arquitectura totalmente diferente de los ordenadores tradicionales de un único procesador. Las máquinas tradicionales basadas en el modelo de Von Neuman tienen un único procesador, la CPU (*Control Process Unit*) que realiza todos los cálculos ejecutando todas las instrucciones de la secuencia programada en el algoritmo. Cualquier CPU realiza más de

cien comandos básicos, incluyendo sumas, restas y desplazamientos entre otros.

Los comandos o instrucciones se ejecutan secuencialmente y sincronizadas con el reloj del sistema. Sin embargo en los sistemas de computación neuronal cada elemento PE sólo puede realizar uno, o como mucho, varios cálculos. La potencia del procesado de los ANN se mide principalmente por el número de interconexiones actualizadas por segundo durante el proceso de entrenamiento o aprendizaje. Sin embargo, las máquinas de Von Neuman se miden por el número de instrucciones que ejecuta por segundo el procesador central CPU.

La arquitectura de las ANN de la organización de los sistemas de procesado en paralelo, es decir, sistemas en los que distintos procesadores están interconectados. No obstante los procesadores son unidades procesadoras simples, diseñadas para la suma de muchas entradas y con un ajuste automático de las conexiones ponderadas.

Sistemas expertos

Los sistemas expertos difieren de la programación tradicional en que la base del conocimiento está separada del motor de inferencia (el método del procesado del conocimiento). Esta característica permite que todo el conocimiento adicional puede ser añadido al sistema sin necesidad de tener que ser reprogramado todo el sistema. Esta técnica requiere que exista una persona experta en un área y que se puedan crear reglas del procesamiento del conocimiento mediante el ajuste de las conexiones ponderadas entre las neuronas de distintas capas de la red.

Mientras que en los Sistemas Expertos el conocimiento se hace explícito en forma de reglas, en la computación neuronal las ANN generan sus propias reglas aprendiendo de los ejemplos que se les muestran en la fase de entrenamiento. El aprendizaje se consigue a través de una regla de aprendizaje que adapta o cambia los pesos de las conexiones en respuesta a los ejemplos de entrada, y que opcionalmente también en respuesta a las salidas deseadas. Esta característica de las ANN es lo que permite decir que las redes neuronales aprenden de la experiencia.

Una característica importante de las ANN es la forma o el modo en que se almacena la información. La memoria o el conocimiento de estas redes está distribuida a lo largo de todas las conexiones ponderadas de la red.

Algunas ANN presentan la característica de ser “asociativas” que significa que para una entrada parcial la red elegirá la entrada más parecida en memoria y generará una salida que corresponda a la entrada completa.

La naturaleza de la memoria de las ANN permite que la red responda adecuadamente cuando se le presenta una entrada incompleta o con ruido. Esta propiedad suele ser referida como la capacidad de “generalización”.

Otra característica de las ANN es la tolerancia a la falta (*Fault Tolerance*). Tolerancia a la falta se refiere al hecho de que en muchas ANN si resultaran destruidos varios elementos procesadores PE, o se alteraran las conexiones el comportamiento de la red sería mínimamente modificado. El comportamiento varía pero el sistema no se descompone o deja de funcionar.

Esta característica se debe a que las ANN tienen la información distribuida a lo largo de toda la red y no está contenida en un único lugar.

2.6.3 Aplicaciones de las Redes Neuronales

Las características especiales de los sistemas de computación neuronal permiten que sea utilizada esta nueva técnica de cálculo en una extensa variedad de aplicaciones.

La computación neuronal provee un acercamiento mayor al reconocimiento y percepción humana que los métodos tradicionales de cálculo. Las redes neuronales artificiales presentan resultados razonables en aplicaciones donde las entradas presentan ruido o las entradas están incompletas. Algunas de las áreas de aplicación de las ANN son las siguientes [18]:

Conversión Texto a Voz: uno de los principales promotores de la computación neuronal en esta área es Terrence Sejnowski. La conversación texto-voz consiste en cambiar los símbolos gráficos de un texto en lenguaje hablado. El sistema de computación neuronal presentado por Sejnowski y Rosemberg, el sistema llamado NetTalk, convierte en texto en fonemas y con la ayuda de un sintetizados de voz (Dectalk) genera voz a partir de un texto escrito.

La ventaja que ofrece la computación neuronal frente a las tecnologías tradicionales en la conversación texto-voz es la propiedad de eliminar la necesidad de programar un complejo conjunto de reglas de pronunciación en el ordenador. A pesar de que el sistema NetTalk ofrece un buen comportamiento, la computación neuronal para este tipo de aplicaciones abre posibilidades de investigación y expectativas de desarrollo comercial.

Procesado Natural del Lenguaje: incluye el estudio de cómo se contruyen las reglas del lenguaje. Los científicos del conocimiento Rumelhart y McClelland han aprendido el tiempo *verbal pass* tense de los verbos en inglés. Las características propias de la computación neuronal como la capacidad de generalizar a partir de datos incompletos y la capacidad de abstraer, permiten al sistema generar buenos diagnósticos para verbos nuevos o verbos desconocidos.

Comprensión de Imágenes: la compresión de imágenes es la transformación de los datos de una imagen a una representación diferente que requiera menos memoria o que se pueda reconstruir una imagen perceptible. Cottrel, Munro y Zisper de la Universidad de San Diego y Pisttburgh han diseñado un sistema de compresión de imágenes utilizando una red neuronal con un factor de compresión de 8:1.

Reconocimiento de Caracteres: es el proceso de interpretación visual y de clasificación de símbolos. Los investigadores de Nestor, Inc. han desarrollado de un sistema de computación neuronal que tras el entrenamiento con un conjunto de tipos de caracteres de letras, es capaz de interpretar un tipo de carácter o letra que no haya visto con anterioridad.

Reconocimiento de Patrones en Imágenes: una aplicación típica es la clasificación de objetivos detectados por un sonar. Existen varias ANN basadas en la popular *Backpropagation* cuyo comportamiento es comparable con el de los operadores humanos. Otra aplicación normal es la inspección industrial.

Problemas de Combinatoria: en este tipo de problemas la solución mediante cálculo tradicional requiere un tiempo de proceso (CPU) que es exponencial con el número de entradas. Un ejemplo es el problema del vendedor; el objetivo es elegir el camino más corto posible que debe realizar el vendedor para cubrir un número limitado de ciudades en un área geográfica específica. Este tipo de problema ha sido abordado con éxito por Hopfield y el resultado de su trabajo ha sido el desarrollo de una ANN que ofrece buenos resultados para este problema de combinatoria.

Procesado de la Señal: en este tipo de aplicación existen tres clases diferentes de procesado de la señal que han sido objeto de las ANN como son la predicción, el modelado de un sistema y el filtrado de ruido.

Predicción: en el mundo real existen muchos fenómenos de los que conocemos su comportamiento a través de una serie temporal de datos o valores. Lapedes y Farber del Laboratorio de Investigación de los Álamos, han demostrado que la red *backpropagation* supera en un orden de magnitud a los métodos de predicción polinómicos y lineales convencionales para las series temporales caóticas.

Modelado de Sistemas: los sistemas lineales son caracterizados por la función de transferencia que no es más que una expresión analítica entre la variable de salida y una variable independiente y sus derivadas. Las ANN también son capaces de aprender una función de transferencia y comportarse correctamente como el sistema lineal que está modelando.

Filtro de Ruido: las ANN también pueden ser utilizadas para eliminar el ruido de una señal. Estas redes son capaces de mantener en un alto grado las estructuras y valores de los filtros tradicionales.

Modelos Económicos y Financieros: una de las aplicaciones más importantes del modelado y pronóstico es la creación de pronósticos económicos como por ejemplo los precios de existencias, la producción de las cosechas, el interés de las cuentas, etc. Las redes neuronales están ofreciendo mejores resultados en los pronósticos financieros que los métodos convencionales.

2.6.4 Entrenamiento de las Redes Neuronales Artificiales

Una de las principales características de las ANN es su capacidad de aprendizaje. El entrenamiento de las ANN muestra algunos paralelismos con el desarrollo intelectual de los seres humanos. No obstante aun cuando parece que se ha conseguido entender el proceso de aprendizaje conviene ser moderado porque el aprendizaje de las ANN está limitado.

El objetivo del entrenamiento de una ANN [19] es conseguir que una aplicación determinada, para un conjunto de entradas produzca el conjunto de salidas deseadas o mínimamente consistentes. El proceso de entrenamiento consiste en la aplicación secuencial de diferentes conjuntos o vectores de entrada para que se ajusten los pesos de las interconexiones según un procedimiento predeterminado. Durante la sesión de entrenamiento los pesos convergen gradualmente hacia los valores que hacen que cada entrada produzca el vector de salida deseado.

Los algoritmos de entrenamiento o los procedimientos de ajuste de los valores de las conexiones de las ANN se pueden clasificar en dos grupos: Supervisado y No Supervisado.

Entrenamiento Supervisado: estos algoritmos requieren el emparejameinto de cada vector de entrada con su correspondiente vector de salida. El entrenamiento consiste en presentar un vector de entrada a la red, calcular la salida de la red, compararla con la salida deseada, y el error o diferencia resultante se utiliza para realimentar la red y cambiar los pesos de acuerdo con un algoritmo que tiende a minimizar el error.

Las parejas de vectores del conjunto de entrenamiento se aplican secuencialmente y de forma cíclica. Se calcula el error y el ajuste de los pesos por cada pareja hasta que el error para el conjunto de entrenamiento entero sea un valor pequeño y aceptable.

Entrenamiento No Supervisado: los sistemas neuronales con entrenamiento supervisado han tenido éxito en muchas aplicaciones, y sin embargo, tienen muchas críticas debido a que desde el punto de vista biológico no son muy lógicos. Resulta difícil creer que existe un mecanismo en el cerebro que compare las salidas deseadas con las salidas reales. En el caso de que exista, ¿de dónde provienen las salidas deseadas?

Los sistemas no supervisados son modelos de aprendizaje más lógicos en los sistemas biológicos. Desarrollados por Kohonen (1984) y otros investigadores, estos sistemas de aprendizaje no supervisado no requieren de un vector de salidas deseadas y por tanto no se realizan comparaciones entre las salidas reales y salidas esperadas. El conjunto de vectores de entrenamiento consiste únicamente en vectores de entrada. El algoritmo de entrenamiento modifica los pesos de la red de forma que produzca vectores de salida consistentes. El proceso de entrenamiento extrae las propiedades estadísticas del conjunto de vectores de entrenamiento y agrupa en clases los vectores similares.

Existe una gran variedad de algoritmos de entrenamiento hoy en día. La gran mayoría de ellos han surgido de la evolución del modelo de aprendizaje no supervisado que propuso Hebb (1949). El modelo propuesto por Hebb se caracteriza por incrementar el valor del peso de la conexión si las dos neuronas unidas son activadas o disparadas. La ley de Hebb se presenta según la ecuación:

$$W_{ij}(n+1) = W_{ij}(n) + \alpha \cdot OUT_i \cdot OUT_j$$

2.7 Predicción de Series Temporales con Redes Neuronales Artificiales

Trabajos previos con ANN han demostrado que éstas tienen habilidades muy potentes de clasificación y reconocimiento de patrones. Inspiradas en los sistemas biológicos, las ANN son capaces de aprender y generalizar de la experiencia pasada. Una de las mayores áreas de aplicación de las ANN es en la Predicción de Series Temporales [13].

La Predicción de Series Temporales ha sido y es uno de los dominios que más esfuerzo se ha dedicado en el campo de la predicción científica. Hoy en día son muchos los datos pasados, almacenados durante años, medidos en diferentes dominios que los científicos usan para poder estudiar qué ocurriría en un futuro próximo. El problema o tarea de predecir series temporales está considerado como la obtención de la relación entre el valor del instante "*t*" y los valores conocidos previos a este (*t-1, t-2, …, t-k*) para poder así establecer una función como se muestra:

$$y_t = f(y_{t-1}, y_{t-2}, \ldots, y_{t-k})$$

donde y_t denota la predicción estimada dada por la función (f).

La predicción en series temporales es una línea de investigación fundamental en el campo de la estadística. El hecho de poder reproducir el comportamiento de un sistema dinámico a partir de medidas discretas (series temporales) de sus variables, posibilita la aplicación de los modelos de predicción basados en series temporales ("*Time Series Based Models*" o TSBM) a una gran cantidad de campos del conocimiento donde los TSBM complementan la modelización física o, incluso, pueden sustituir a la modelización física cuando esta se hace demasiada compleja. Para que el análisis de una serie temporal conduzca a conclusiones acertadas no basta con utilizar las técnicas apropiadas, sino que resulta imprescindible que esos datos son comparables, y estos no lo serán si no son homogéneos. Por ejemplo, si cada año cambia la metodología de observación, se cambian las definiciones y por lo tanto se modifica la población de referencia. Entonces, el resultado será una serie temporal compuesta por un conjunto de valores no comparables porque se vuelven muy heterogéneos. Esta homogeneidad se puede perder, de una forma natural con el transcurso del tiempo, por lo tanto, cuando las series son muy largas no hay garantía de que los datos iniciales y finales sean comparables.

Sin embargo, esta necesidad de que las series no sean largas, para que sus datos no se pierdan la deseable homogeneidad, entra en contradicción con el objetivo más elemental de la estadística, que es el de detectar regularidades en los fenómenos de masas. La planificación nos exige prever los sucesos del futuro que puedan ocurrir. La previsión, a su vez, se suele basar en lo que ha ocurrido en el pasado. Se tiene pues un nuevo tipo de inferencia estadística que se hace acerca del futuro de alguna variable o compuesto de variables basándose en sucesos pasados. La técnica más importante para hacer inferencias sobre el futuro basándose en lo ocurrido en el pasado, es el análisis de series temporales.

Los pasos en el análisis de las series temporales son:

- Detectar puntos que se escapan de lo normal.
- Detectar tendencias cuando, a largo plazo, hay un incremento o decremento de los datos.
- Variación estacional, el comportamiento de la variable esta influenciada por periodos como estaciones, días, trimestres, años, …

Una vez llevadas a cabo las acciones anteriores, el siguiente paso consistiría en determinar si la secuencia de valores es completamente aleatoria o si, por el contrario, se puede encontrar algún patrón a lo largo del tiempo, pues sólo en este caso podremos seguir con el análisis. La metodología tradicional para el estudio de series temporal resulta sencilla de comprender, y fundamentalmente se basa en descomponer las series en varias partes: tendencia, variación cíclica, variación estacional o periódica, y otras fluctuaciones irregulares:

- **Tendencias**: es la dirección general de la variable en el periodo de observación, es decir el cambio a largo plazo de la media de la serie.
- **Variación cíclica**: es una componente de la serie que recoge oscilaciones periódicas de amplitud superior a un año. Estas oscilaciones periódicas no son regulares y se presentan en los fenómenos económicos cuando se dan de forma alternativa etapas de prosperidad o de depresión.
- **Periodicidad**: es una componente de la serie que recoge oscilaciones periódicas de amplitud diferentes a un año, mes, semanas, etc. Estas oscilaciones periódicas no son regulares y se presentan en los fenómenos económicos cuando se dan de forma alternativa etapas de prosperidad o de depresión.
- **Otras fluctuaciones irregulares**: después de extraer de la serie la tendencia y las variaciones cíclicas y estacionales, nos quedará una serie de valores residuales, que pueden ser o no aleatorios. Volvemos a estar como en el punto de partida, pues ahora también nos interesa determinar si esa secuencia temporal de valores residuales puede o no corresponderse con un patrón puramente aleatorio.

Lo que se pretende con una serie es tanto describir como predecir el comportamiento de un fenómeno que cambia en el tiempo. El principal objetivo

de esta tesis es el desarrollo de una herramienta automática basada en técnicas de IA, que lleve a cabo la Predicción de Series Temporales, haciendo prácticamente innecesaria la intervención de ningún experto, ya que dicha herramienta aprenderá a partir de la información contenida en los propios datos, es decir, los valores de dicha serie temporal.

2.8 Introducción máquinas paralelas

Tradicionalmente, la simulación numérica de sistemas complejos como dinámica de fluidos, clima, circuitos eléctricos, reacciones químicas, modelos ambientales y procesos de manufacturación, han impulsado el desarrollo de computadores cada vez más potentes. Hoy en día, estas máquinas están siendo promovidas por aplicaciones comerciales que requieren procesar grandes cantidades de datos. Entre ellas encontramos realidad virtual, video conferencias, bases de datos paralelas, diagnóstico médico asistido por computadores.

La eficiencia de un computador depende directamente del tiempo requerido para ejecutar una instrucción básica y del número de instrucciones básicas que pueden ser ejecutadas concurrentemente. Esta eficiencia puede ser incrementada por avances en la arquitectura y por avances tecnológicos. Avances en la arquitectura incrementan la cantidad de trabajo que se puede realizar por ciclo de instrucción: memoria bit-paralela, aritmética bit-paralela, memoria caché, memoria intercalada, múltiples unidades funcionales, *lookahead* de instrucciones, *pipelining* de instrucciones, unidades funcionales *pipelined* y *pipelining* de datos. Una vez incorporados estos avances, mejorar la eficiencia de un procesador implica reducir el tiempo de los ciclos: avances tecnológicos.

Hace un par de décadas, los microprocesadores no incluían la mayoría de los avances de arquitectura que ya estaban presentes en los supercomputadores. Esto ha causado que en el último tiempo el adelanto visto en los microprocesadores haya sido significativamente más notable que el de otros tipos de procesadores: supercomputadores, *mainframes* y minicomputadoras. En la figura 8 se puede apreciar el crecimiento en la eficiencia para los minicomputadoras, mainframes y supercomputadores ha estado por debajo del 20% por año, mientras que para los microprocesadores ha sido de un 35% anual en promedio.

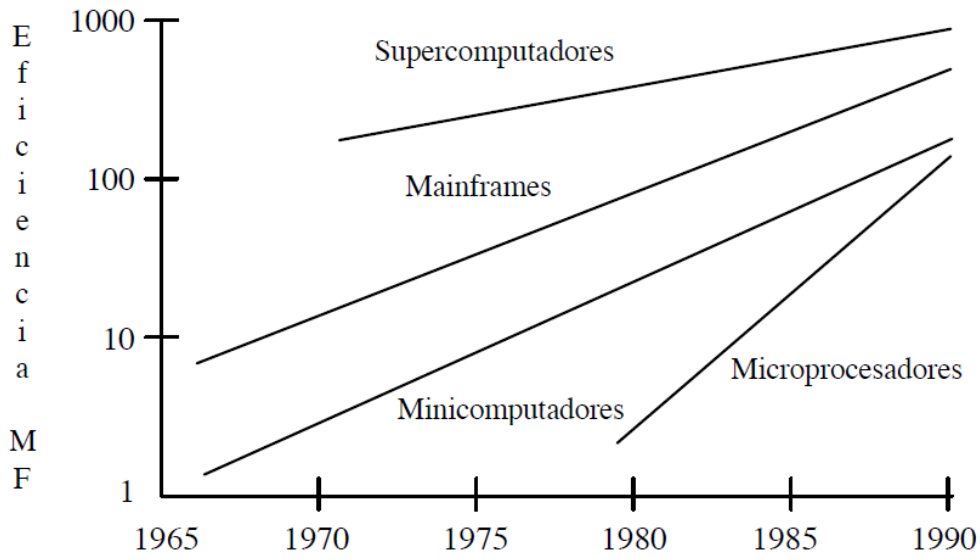

Figura 8 - Crecimiento en la eficiencia de los computadores en megaflops

El tiempo para ejecutar una operación básica definitivamente depende del tiempo de los ciclos del procesador, es decir, el tiempo para ejecutar la operación más básica. Sin embargo, estos tiempos están decreciendo lentamente y parece que están alcanzando límites físicos como la velocidad de la luz (figura 9).

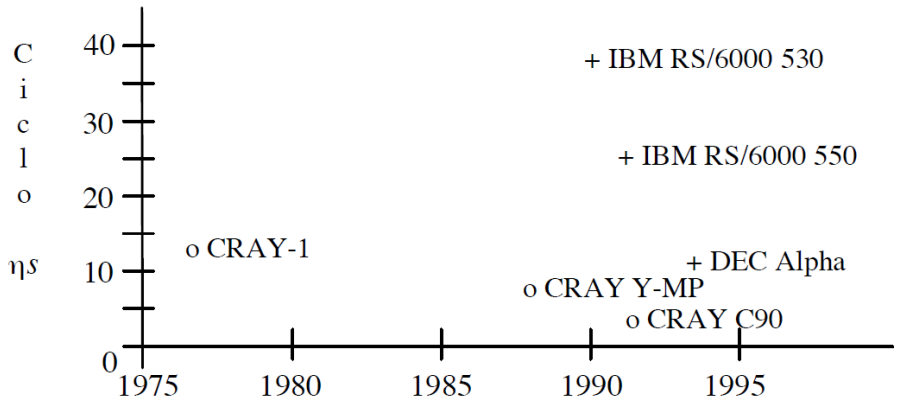

Figura 9 - Tendencias en el tiempo de los ciclos (o = supercomputadores, + = microcomputadores, RISC)

Dadas estas dificultades en mejorar la eficiencia de un procesador, la convergencia relativa en eficiencia entre microprocesadores y los supercomputadores tradiciones, y el relativo bajo costo de los microprocesadores, ha permitido el desarrollo de computadores paralelos viables comercialmente con decenas, cientos y hasta miles de microprocesadores.

Un **computador paralelo** es un conjunto de procesadores capaces de cooperar en la solución de un problema. Esta definición incluye supercomputadores con cientos de procesadores, redes de estaciones de trabajo y máquinas con múltiples procesadores.

2.8.1 Clasificación de los sistemas paralelos

Probablemente la clasificación más popular de computadores sea la clasificación de *Flynn* [20][21]. Esta taxonomía de las arquitecturas está basada en la clasificación atendiendo al flujo de datos e instrucciones en un sistema. Un flujo de instrucciones es el conjunto de instrucciones secuenciales que son ejecutadas por un único procesador, y un flujo de datos es el flujo secuencial de datos requeridos por el flujo de instrucciones. Con estas consideraciones, Flynn clasifica los sistemas en cuatro categorías:

SISD (*Single Instruction stream, Single Data stream*)[41]**.** Flujo único de instrucciones y flujo único de datos. Este es el concepto de arquitectura serie de *Von Neumann* donde, en cualquier momento, sólo se está ejecutando una única instrucción. A menudo a los SISD se les conoce como computadores serie escalares. Todas las máquinas SISD poseen un registro simple que se llama **contador de programa** que asegura la ejecución en serie del programa. Conforme se van leyendo las instrucciones de la memoria, el contador de programa se actualiza para que apunte a la siguiente instrucción a procesar en serie. Prácticamente ningún computador puramente SISD se fabrica hoy en día ya que la mayoría de los procesadores modernos incorporan algún grado de paralelización como es la segmentación de instrucciones o la posibilidad de lanzar dos instrucciones a un mismo tiempo (superescalares).

MISD (*Multiple Instruction stream, Single Data stream*). Flujo múltiple de instrucciones y único flujo de datos. Esto significa que varias instrucciones actúan sobre el mismo y único flujo de datos. Esto significa que varias instrucciones actúan sobre el mismo y único trozo de datos. Este tipo de máquinas se pueden interpretar de dos maneras. Una es considerar la clase de máquinas que requerirían que unidades de procesamiento diferentes recibieran instrucciones distintas operando sobre los mismos datos. Esta clase de arquitectura ha sido clasificada por numerosos arquitectitos de computadores como impracticable o imposible, y en estos momentos no existen ejemplos que funcionen siguiendo este modelo. Otra forma de interpretar los MISD es como una clase de máquinas donde un mismo flujo de datos fluye a través de numerosas unidades procesadoras. Arquitecturas altamente segmentadas, como los *arrays* sistólicos o los procesadores vectoriales, son clasificados a menudo bajo este tipo de máquinas. Las arquitecturas segmentadas, o encauzadas, realizan el procesamiento vectorial a través de una serie de etapas, cada una ejecutando una función particular produciendo un resultado intermedio. La razón por la cual dichas arquitecturas son clasificadas como MISD es que los elementos de un vector pueden ser considerados como pertenecientes al mismo dato, y todas las etapas del cauce representan múltiples instrucciones que son aplicadas sobre ese vector.

SIMD (*Single Instruction stream, Multiple Data stream*). Flujo de instrucción simple y flujo de datos múltiple. Esto significa que una única instrucción es aplicada sobre diferentes al mismo tiempo. En las máquinas de este tipo, varias unidades de procesado diferentes son invocadas por una única unidad de control. Al igual que las MISD, las SIMD soportan procesamiento vectorial (matricial) asignando cada elemento del vector a una unidad funcional diferente para procesamiento concurrente. Por esta facilidad en la paralelización de vectores de datos se les llama también procesadores matriciales.

MIMD (*Multiple Instruction stream, Multiple Data stream*). Flujo de instrucciones múltiple y flujo de datos múltiple. Son máquinas que poseen varias unidades procesadoras en las cuales se pueden realizar múltiples instrucciones sobre datos diferentes de forma simultánea. Las MIMD son las más complejas, pero son también las que potencialmente ofrecen una mayor eficiencia en la ejecución concurrente o paralela. Aquí la concurrencia implica que no sólo hay varios procesadores operando simultáneamente, sino que además hay varios programas (procesos) ejecutándose también al mismo tiempo.

La figura 10 muestra los esquemas de estos cuatro tipos de máquinas clasificadas por los flujos de instrucciones y datos.

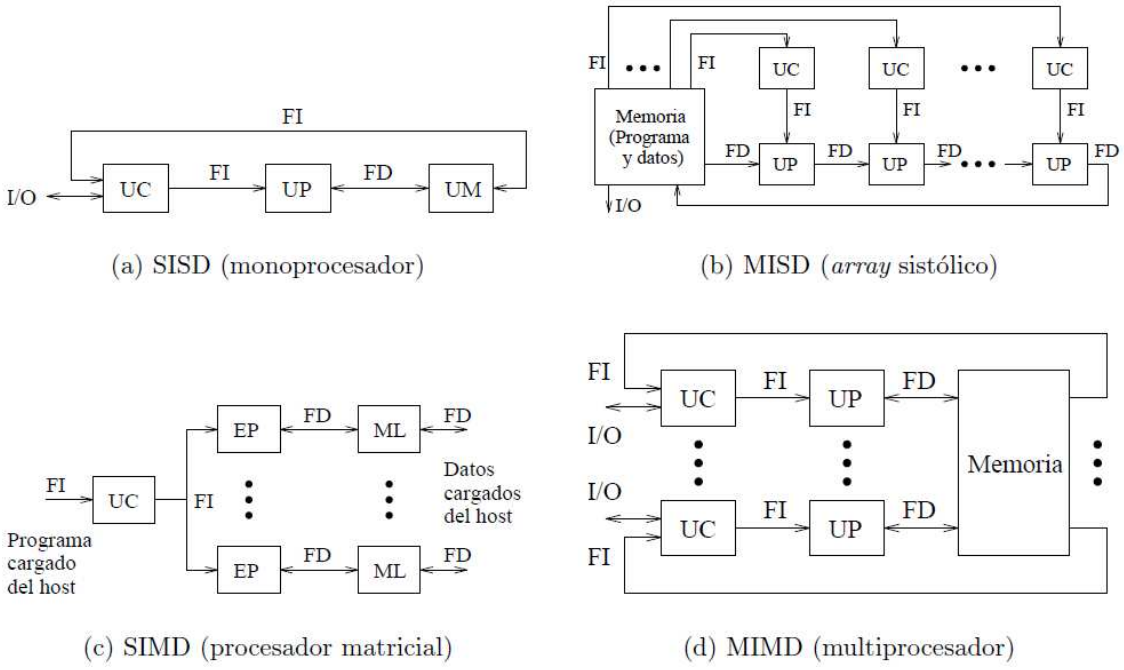

Figura 10 - Clasificación de Flynn de las arquitecturas de computadores. (UC=Unidad de Control, UP=Unidad de Procesamiento, UM=Unidad de Memoria, EP= Elemento de Proceso, ML=Memoria Local, FI=Flujo de Instrucciones, FD=Flujo de Datos)

2.8.2 Otras Clasificaciones

La clasificación de Flynn ha demostrado funcionar bastante bien para la tipificación de sistemas, y se ha venido usando desde décadas por la mayoría de los arquitectos de computadores. Sin embargo, los avances en tecnología y diferentes topologías, han llevado a sistemas que no son tan fáciles de clasificar dentro de los 4 tipos de Flynn [22][23]. Por ejemplo, los procesadores vectoriales no encajan adecuadamente en esta clasificación, ni tampoco las arquitecturas híbridas. Para solucionar esto se han propuesto otras clasificaciones, donde los tipos SIMD y MIMD de Flynn se suelen conservar, pero que sin duda no han tenido el éxito de la de Flynn.

La figura 11 muestra una taxonomía ampliada que incluye alguno de los avances en arquitecturas de computadores en los últimos años. No obstante, tampoco pretende ser una caracterización completa de todas las arquitecturas paralelas existentes.

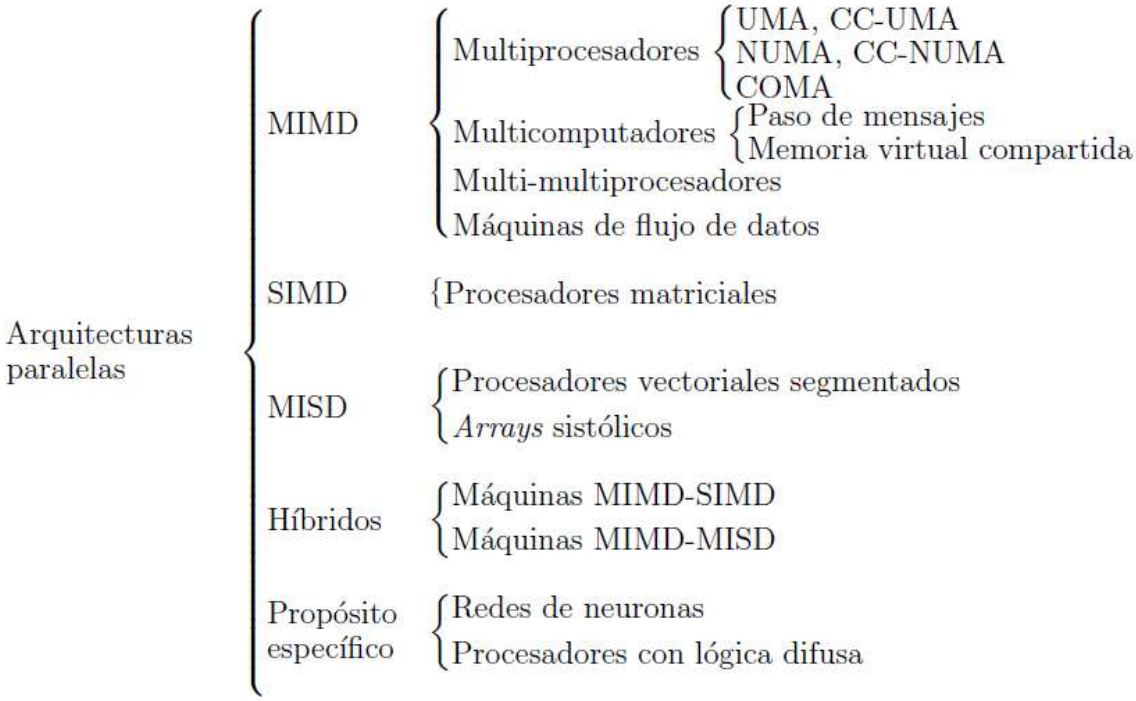

Figura 11 - Clasificación de arquitecturas paralelas

Tal y como se ve en la figura, los de tipo MIMD pueden a su vez ser subdivididos en multiprocesadores, multicomputadores, multi-multiprocesadores y máquinas de flujo de datos. Incluso los multiprocesadores pueden ser subdivididos en NUMA, UMA y COMA según el modelo de memoria compartida. El tipo SIMD quedaría con los procesadores matriciales y el MISD se subdividiría en procesadores vectoriales y en *arrays* sistólicos. Se han añadido dos tipos más que son el híbrido y los de aplicación específica.

Multiprocesadores

Un multiprocesador se puede ver como un computador paralelo compuesto por varios procesadores interconectados que pueden compartir un mismo sistema de memoria. Los procesadores se pueden configurar para que ejecute cada uno una parte de un programa o varios programas al mismo tiempo. Un diagrama de bloques de esta arquitectura se muestra en la figura 12. Tal y como se muestra en la figura, que se corresponde a un tipo particular de multiprocesador que se verá más adelante, un multiprocesador está generalmente formado por *n* procesadores y *m* módulos de memoria. A los procesadores los llamamos $P_1, P_2, \ldots, P_n$ y a las memorias $M_1, M_2, \ldots, M_n$. La red de interconexión conecta cada procesador a un subconjunto de los módulos de memoria.

Dado que los multiprocesadores comparten los diferentes módulos de memoria, pudiendo acceder varios procesadores a un mismo módulo, a los multiprocesadores también se les llama *sistemas de memoria compartida*. Dependiendo de la forma en que los procesadores comparten la memoria, podemos hacer una subdivisión de los multiprocesadores:

UMA (*Uniform Memory Access*). En un modelo de Memoria de Acceso Uniforme, la memoria física está uniformemente compartida por todos los procesadores. Esto quiere decir que todos los procesadores tienen el mismo tiempo de acceso a todas las palabras de memoria. Cada procesador puede tener su caché privada, y los periféricos son también compartido de alguna manera.

A estos computadores se les puede llamar *sistemas fuertemente acoplados* dado el alto grado de compartición de recursos. La red de interconexión toma la forma de bus común, conmutador cruzado, o una red multietapa.

Cuando todos los procesadores tienen el mismo acceso a todos los periféricos, el sistema se llama multiprocesador *simétrico*. En este caso, todos los procesadores tienen la misma capacidad para ejecutar programas, tal como el kernel o las rutinas de servicio E/S. En un multiprocesador *asimétrico*, sólo un subconjunto de los procesadores pueden ejecutar programas. A los que pueden, o al que puede ya que muchas veces es sólo uno, se le llama *maestro*. Al resto de procesadores se les llama *procesadores adheridos* (*attached processors*). La figura 12 muestra el modelo UMA de un multiprocesador.

Es frecuente encontrar arquitecturas de acceso uniforme que además tiene coherencia de caché, a estos sistemas se les suele llamar **CC-UMA** (*Cache-Coherent Uniform Memory Access*).

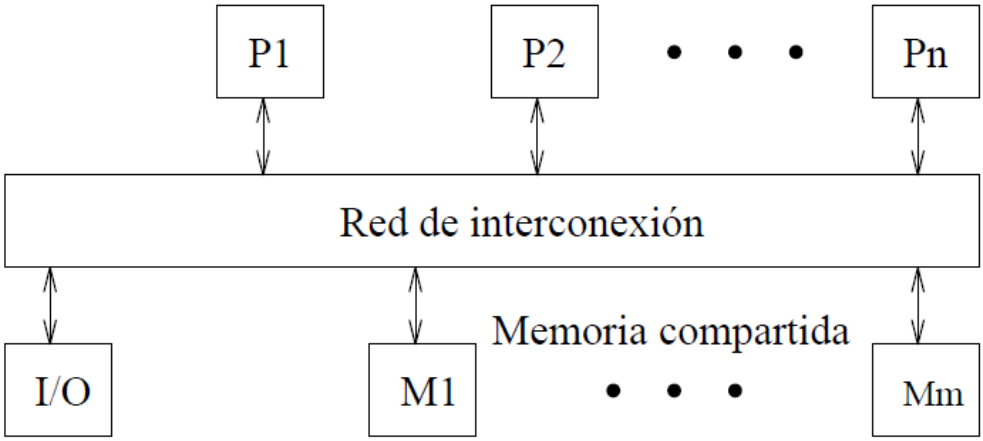

Figura 12 - El modelo UMA de multiprocesador

NUMA (*Non-Uniform Memory Access*). Un multiprocesador de tipo NUMA es un sistema de memoria compartida donde el tiempo de acceso varía según el lugar donde se encuentre localizado el acceso. La figura 13 muestra una posible configuración de tipo NUMA, donde toda la memoria es compartida pero local a cada módulo procesador. Otras posibles configuraciones incluyen los sistemas basados en agrupaciones (clusters) de sistemas como el de la figura que se comunican a través de otra red de comunicación que puede incluir una memoria compartida global.

La ventaja de estos sistemas es que el acceso a la memoria local es más rápido que en los UMA aunque un acceso a la memoria no local es más lento. Lo que se intenta es que la memoria utilizada por los procesos que ejecuta cada procesador, se encuentre en la memoria de dicho procesador para que los accesos sean lo más locales posible.

Aparte de esto, se puede añadir al sistema una memoria de acceso global. En este caso se dan tres posibles patrones de acceso. El más rápido es el acceso a memoria local. Le sigue el acceso a memoria global. El más lento es el acceso a la memoria del resto de módulos.

Al igual que hay sistemas de tipo CC-UMA, también existe el modelo de acceso a memoria no uniforme con coherencia de caché **CC-NUMA** (*Cache-Coherent Non-Uniform Memory Access*) que consiste en memoria compartida distribuida y directorios de caché.

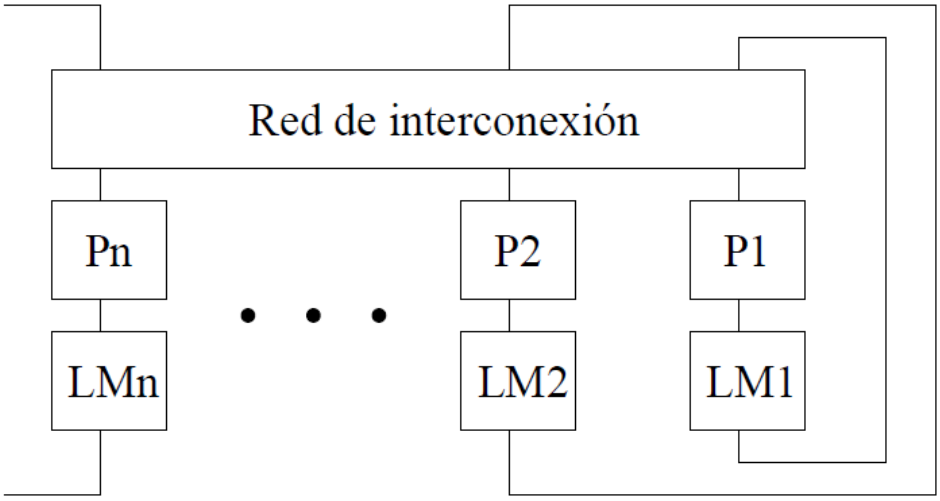

Figura 13 - Modelo NUMA de multiprocesador

COMA (*Cache Only Memory Access*). Un multiprocesador que solo use caché como memoria es considerado de tipo COMA. La figura 14 muestra el modelo COMA de multiprocesador. En realidad, el modelo COMA es un caso especial del NUMA donde las memorias distribuidas se convierten en cachés. No hay jerarquía de memoria en cada módulo procesador. Todas las cachés forman un mismo espacio global de direcciones. El acceso a las cachés remotas se realiza a través de los directorios distribuidos de las cachés. Dependiendo de la red de interconexión empleada, se pueden utilizar jerarquías en los directorios para ayudar en la localización de copias de bloques de caché. El emplazamiento inicial de datos no es crítico puesto que el dato acabará estando en el lugar que se use más.

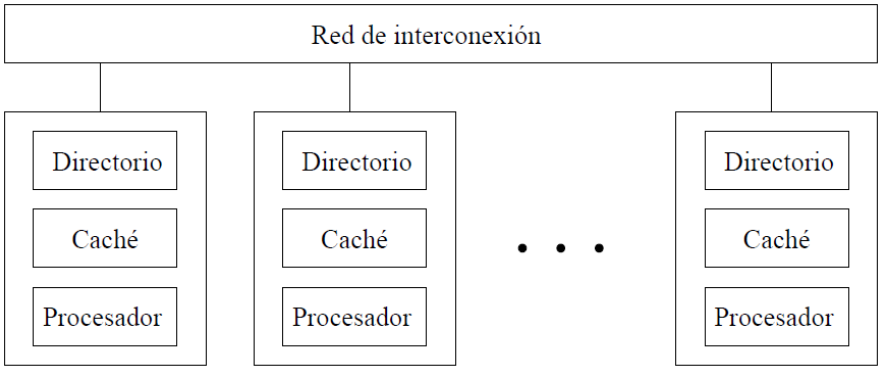

Figura 14 - El modelo COMA de multiprocesador

Multicomputadores

Un multicomputador se puede ver como un computador paralelo en el cual cada procesador tiene su propia memoria local. La memoria del sistema se encuentra distribuida entre todos los procesadores y cada procesador sólo puede direccionar su memoria local; para acceder a las memorias de los demás procesadores debe hacerlo por *paso de mensajes*. Esto significa que un procesador tiene acceso directo sólo a su memoria local, siendo indirecto el acceso al resto de memorias del resto de procesadores. Este acceso local y privado a la memoria es lo que diferencia los multicomputadores de los multiprocesadores.

El diagrama de bloques de un multicomputador coincide con el visto en la figura 14 que corresponde a un modelo NUMA de procesador, la diferencia viene dada porque la red de interconexión no permite un acceso directo entre memorias, sino que la comunicación se realiza por paso de mensajes.

La transferencia de datos se realiza a través de la red de interconexión que conecta un subconjunto de procesadores con otro subconjunto. La transferencia de unos procesadores a otros se realiza por tanto por múltiples transferencias entre procesadores conectados dependiendo de cómo esté establecida la red.

Dado que la memoria está distribuida entre los diferentes elementos de proceso, a estos sistemas se les llama *distribuidos* aunque no hay que olvidar que pueden haber sistemas que tengan la memoria distribuida pero compartida y por lo tanto no ser multicomputadores. Además, y dado que se explota mucho la localidad, a estos sistemas se les llama *débilmente acoplados*, ya que los módulos funcionan de forma casi independiente unos de otros.

Multicomputadores con memoria virtual compartida

En un multicomputador, un proceso de usuario puede construir un espacio global de direccionamiento virtual. El acceso a dicho espacio global de direccionamiento se puede realizar por software mediante un paso de mensajes explícito. En las bibliotecas de paso de mensajes hay siempre rutinas que permiten a los procesos aceptar mensajes de otros procesos, con lo que cada proceso puede servir datos de su espacio virtual a otros procesos. Una lectura se realiza mediante el envío de una petición al proceso que contiene el objeto. La petición por medio del paso de mensajes puede quedar oculta al usuario, ya que puede haber sido generada por el compilador que tradujo el código de acceso a una variable compartida.

De esta manera el usuario se encuentra programando un sistema aparentemente basado en memoria compartida cuando en realidad se trata de un sistema basado en el paso de mensajes. A este tipo de sistemas se les llama multicomputadores con memoria virtual compartida.

Otra forma de tener un espacio de memoria virtual compartido es mediante el uso de páginas. En estos sistemas una colección de procesos tiene una región de direcciones compartidas pero, para cada proceso, sólo las páginas que son locales son accesibles de forma directa. Si se produce un acceso a una página remota, entonces se genera un fallo de página y el sistema operativo inicia una secuencia de pasos de mensaje para transferir la página y ponerla en el espacio de direcciones del usuario.

Máquinas de flujo de datos

Hay dos formas de procesar la información, una es mediante la ejecución en serie de una lista de comandos y la otra es la ejecución de un comando demandando por los datos disponibles. La primera forma empezó con la arquitectura Von Neumann donde un programa almacenaba las órdenes a ejecutar, sucesivas modificaciones, etc., han convertido esta sencilla arquitectura en los multiprocesores para permitir paralelismo.

La segunda forma de ver el procesamiento de datos quizá es algo menos directa, pero desde el punto de vista de la paralelización resulta mucho más interesante puesto que las instrucciones que se ejecutan en el momento tienen los datos necesarios para ello, y naturalmente se debería poder ejecutar todas las instrucciones demandadas en un mismo tiempo. Hay algunos lenguajes que se adaptan a este tipo de arquitectura comandada por datos como son el Prolog, ADA, etc., es decir, lenguajes que exploten de una y otra manera la concurrencia de instrucciones.

En una arquitectura de flujo de datos una instrucción está lista para su ejecución cuando los datos que necesita están disponibles. La disponibilidad de los datos se consigue por la canalización de los resultados de las instrucciones ejecutadas con anterioridad a los operandos de las instrucciones que esperan. Esta canalización forma un flujo de datos que van disparando las instrucciones a ejecutar. Para esto se evita la ejecución de instrucciones basadas en *contador de programa* que es la base de la arquitectura Von Neumann.

Las instrucciones en un flujo de datos son puramente autocontenidas; es decir, no direccionan variables en una memoria compartida global, sino que llevan los valores de las variables en ellas mismas. En una máquina de este tipo, la ejecución de una instrucción no afecta a otras que estén listas para su ejecución. De esta manera, varias instrucciones pueden ser ejecutadas simultáneamente lo que lleva a la posibilidad de un alto grado de concurrencia y paralelización.

La figura 15 muestra el diagrama de bloques de una máquina de flujo de datos. Las instrucciones, junto con sus operandos, se encuentran almacenados en la memoria de datos e instrucciones (D/I). Cuando una instrucción está lista para ser ejecutada, se envía a uno de los elementos de proceso (EP) a través de la red de arbitraje. Cada EP es un procesador simple con memoria local limitada. El EP, después de procesar la instrucción, envía el resultado a su destino a través de la red de distribución.

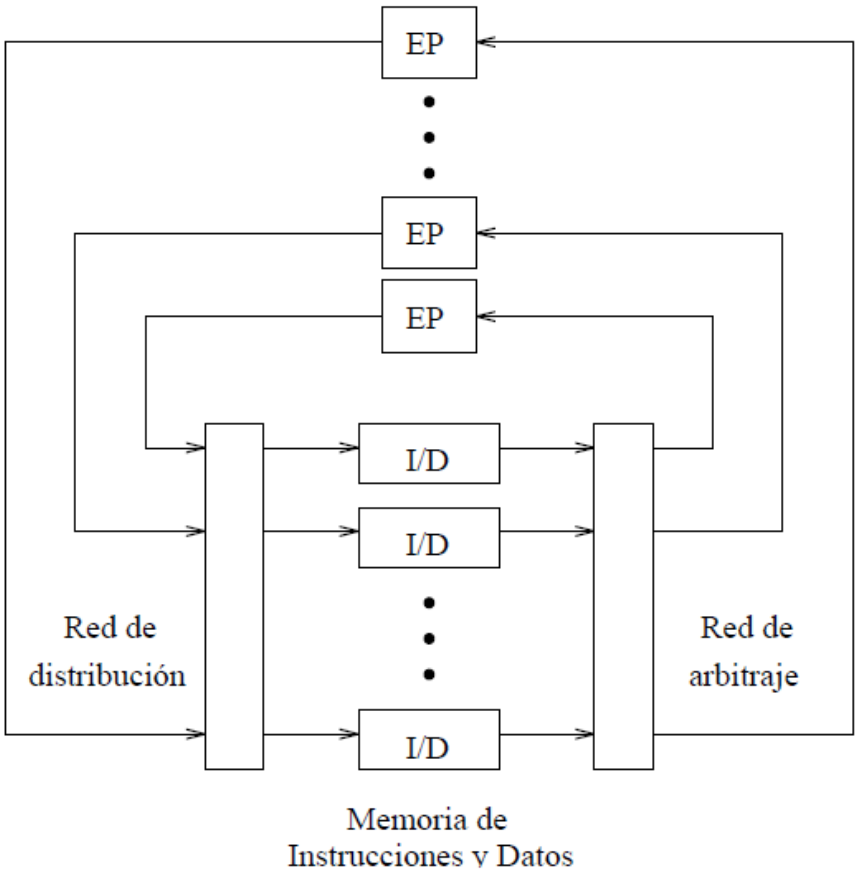

Figura 15 - Diagrama de bloques de una máquina de flujo de datos

Procesadores matriciales

Esta arquitectura es la representativa del tipo SIMD, es decir, hay una sola instrucción que opera concurrentemente sobre múltiples datos.

Un procesador matricial consiste en un conjunto de elementos de proceso y un procesador escalar que operan bajo una unidad de control. La unidad de control busca y decodifica las instrucciones de la memoria central y las manda bien al procesador escalar o bien a los nodos procesadores dependiendo del tipo de instrucción. La instrucción que ejecutan los nodos procesadores es la misma simultáneamente, los datos serán los de cada memoria de procesador y por tanto serán diferentes. Por todo esto, un procesador matricial sólo requiere un único programa para controlar todas las unidades de proceso.

La idea de utilización de los procesadores matriciales es explotar el paralelismo en los datos de un problema más que paralelizar la secuencia de ejecución de las instrucciones. El problema se paraleliza dividiendo los datos en particiones sobre las que se pueden realizar las mismas operaciones. Un tipo de datos

altamente particionable es el formado por vectores y matrices, por eso a estos procesadores se les llama matriciales.

Procesadores vectoriales

Un procesador vectorial ejecuta de forma segmentada instrucciones sobre vectores. La diferencia con los matriciales es que mientras los matriciales son comandados por las instrucciones, los vectoriales son comandados por flujos de datos continuos. A este tipo se le considera MISD puesto que varias instrucciones son ejecutadas sobre un mismo dato (el vector), si bien es una consideración algo confusa aunque aceptada de forma mayoritaria.

Arrays sistólicos

Otro tipo de máquinas que se suelen considerar MISD son los *arrays* sistólicos. En un *array* sistólico hay un gran número de elementos de proceso (EPs) idénticos con una limitada memoria local. Los EPs están colocados en forma de matriz (*array*) de manera que sólo están permitidas las conexiones con los EPs vecinos. Por lo tanto, todos los procesadores que se encuentran organizados en una estructura segmentada de forma lineal o matricial. Los datos fluyen de unos EPs a sus vecinos a cada ciclo de reloj, y durante ese ciclo de reloj, o varios, los elementos de proceso realizan una operación sencilla. El adjetivo sistólico viene precisamente del hecho de que todos los procesadores vienen sincronizados por un único reloj que hace de “corazón” que hace moverse a la máquina.

Arquitecturas híbridas

Se han visto dos formas de explotar el paralelismo. Por un lado estaba la paralelización de código que se consigue con las máquinas de tipo MIMD, y por otro lado, estaba la paralelización de los datos conseguida con arquitectura SIMD y MISD. En la práctica, el mayor beneficio en paralelismo viene de la paralelización de los datos. Esto es debido a que el paralelismo de los datos explota el paralelismo en proporción a la cantidad de los datos que forman el cálculo a realizar. Sin embargo, muchas veces resulta imposible explotar el paralelismo inherente en los datos del programa y se hace necesario utilizar tanto el paralelismo de control como el de datos. Por lo tanto, procesadores que tienen características de MIMD y SIMD (o MISD) a un tiempo, pueden resolver de forma efectiva un elevado rango de problemas.

Arquitecturas específicas

Las arquitecturas específicas son muchas veces conocidas también con el nombre de arquitecturas VLSI ya que muchas llevan consigo la elaboración de circuitos específicos con una alta escala de integración.

Un ejemplo de arquitectura de propósito específico son las redes neuronales (ANN de *Artificial Neural Network*). Las ANN consisten en un elevado número de elementos de proceso muy simples que operan en paralelo. Estas arquitecturas se pueden utilizar para resolver el tipo de problemas que a un

humano le resultan fáciles y a una máquina tan difícil, como el reconocimiento de patrones, comprensión del lenguaje, etc. La diferencia con las arquitecturas clásicas es la forma en que se programa; mientras en un arquitectura Von Neumann se aplica un programa o algoritmo para resolver un problema, una red de neuronas aprende a fuerza de aplicarle patrones de comportamiento.

La idea es la misma que en el cerebro humano. Cada elemento de proceso es como una neurona con numerosas entradas provenientes de otros elementos de proceso y una única salida que va a otras neuronas o a la salida del sistema. Dependiendo de los estímulos recibidos por las entradas a la neurona la salida se activará o no dependiendo de una función de activación. Este esquema permite dos cosas, por un lado que la red realice una determinada función según el umbral de activación interno de cada neurona, y por otro, va a permitir que pueda programarse la red mediante la técnica de ensayo-error.

Otro ejemplo de dispositivo de uso específico son los procesadores basado en *lógica difusa*. Estos procesadores tienen que ver con los principios del razonamiento aproximado. La lógica difusa intenta tratar con la complejidad de los procesos humanos eludiendo los inconvenientes asociados a la lógica de dos valores clásica.

2.9 Diseño de Algoritmo Paralelos

El diseño involucra cuatro etapas [24] las cuales se presentan como secuenciales pero que en la práctica no lo son:

1. **Partición**: El cómputo y los datos sobre los cuales se opera se descomponen en tareas. Se ignoran aspectos como el número de procesadores de la máquina a usar y se concentra en la atención en explorar oportunidades de paralelismo.

2. **Comunicación**: Se determina la comunicación requerida para coordinar las tareas. Se definen estructuras y algoritmos de comunicación.

3. **Agrupación:** El resultado de las dos etapas anteriores es evaluado en términos de eficiencia y costos de implementación. De ser necesario, se agrupan tareas pequeñas en tareas más grandes.

4. **Asignación:** Cada tarea es asignada a un procesador tratando de maximizar la utilización de los procesadores y de reducir el costo de comunicación. La asignación puede ser estática (se establece antes de la ejecución del programa) o a tiempo de ejecución mediante algoritmos de balanceo de carga.

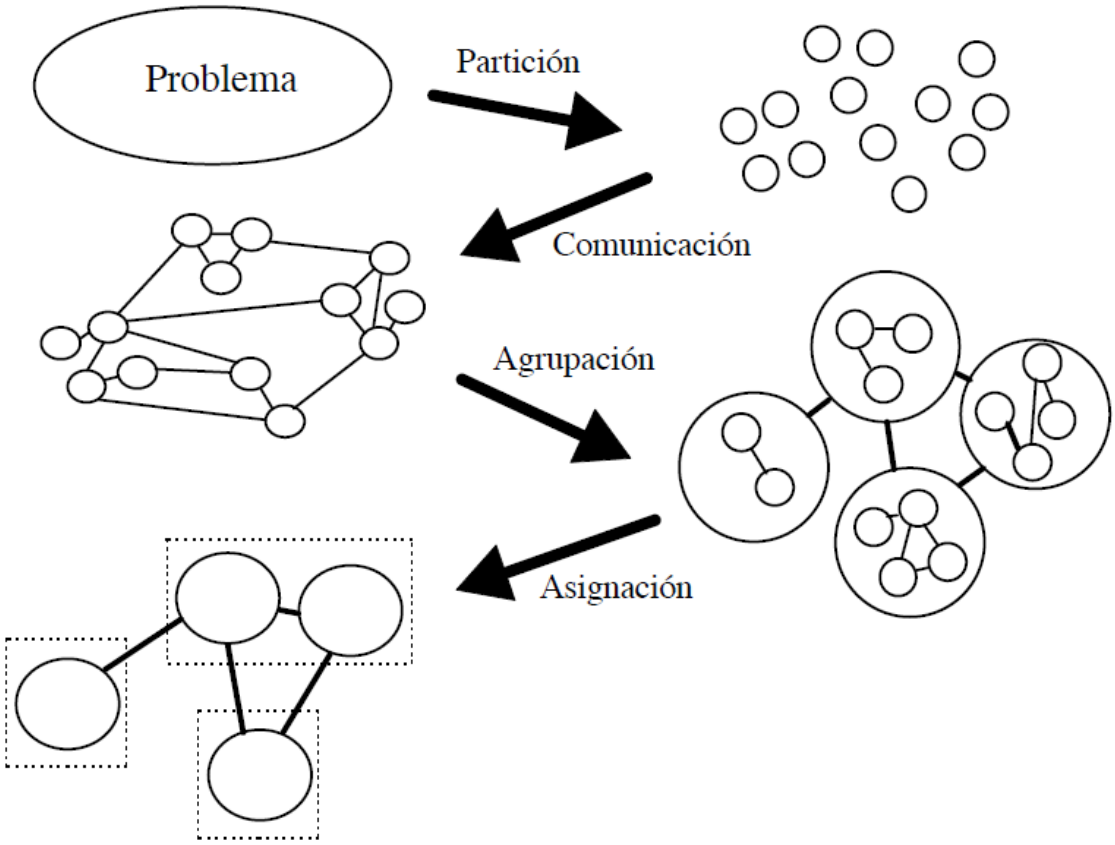

Figura 16 - Etapas en el diseño de algoritmos paralelos

2.9.1 Partición

En la etapa de partición se buscan oportunidades de paralelismo y se trata de subdividir el problema lo más finamente posible, es decir; que la **granularidad** sea **fina**. Evaluaciones futuras podrán llevar a aglomerar tareas y descartar ciertas posibilidades de paralelismo. Una buena partición divide tanto los cómputos como los datos. Hay dos formas de proceder con la descomposición.

Descomposición del dominio: el centro de atención son los datos. Se determina la partición apropiada de los datos y luego se trabaja en los cómputos asociados con los datos.

Descomposición funcional: es el enfoque alternativo al anterior. Primero se descomponen los cómputos y luego se ocupa de los datos.

Estas técnicas son complementarias y pueden ser aplicadas a diferentes componentes de un problema e inclusive al mismo problema para obtener algoritmos paralelos alternativos.

2.9.2 Comunicación

Las tareas definidas en la etapa anterior, en general, pueden correr concurrentemente pero no independientemente. Datos deben ser transferidos o compartidos entre tareas y estos es lo que se denomina la fase de comunicación.

La comunicación requerida por un algoritmo puede ser definida en dos fases. Primero se definen los canales que conectan las tareas que requieren datos con las que los poseen. Segundo se especifica la información o mensajes que deben ser enviados y recibidos en estos canales.
Dependiendo del tipo de máquina en que se implementará el algoritmo, memoria distribuida o memoria compartida, la forma de atacar la comunicación entre tareas varía.

En ambientes de memoria distribuida, cada tarea tiene una identificación única y las tareas interactúan enviando y recibiendo mensajes hacia y desde tareas específicas. Las librerías más conocidas para implementar el pase de mensajes en ambientes de memoria distribuida son: MPI (*Message Passing Interface*) y PVM (*Parallel Virtual Machine*).

En ambientes de memoria compartida no existe la noción de pertenencia y el envío de datos no se da como tal. Todas las tareas comparten la misma memoria. Semáforos, barreras y otros mecanismos de sincronización son usados para controlar el acceso a la memoria compartida y coordinar las tareas.

En esta etapa hay que tener en cuenta los siguientes aspectos:

1) Todas las tareas deben efectuar aproximadamente el mismo número de operaciones de comunicación. Si esto no se da, es muy probable que el algoritmo no sea extensible a problemas mayores ya que habrá cuellos de botella.
2) La comunicación entre tareas debe ser tan pequeña como sea posible.
3) Las operaciones de comunicación deben poder proceder concurrentemente.
4) Los cómputos de diferentes tareas deben poder proceder concurrentemente.

2.9.3 Agrupación

En las dos etapas anteriores se particionó el problema y se determinaron los requerimientos de comunicación. El algoritmo resultante es aún abstracto en el sentido de que no se tomó en cuenta la máquina sobre el cual correrá. En esta etapa se va de lo abstracto a lo concreto y se revisa el algoritmo obtenido tratando de producir un algoritmo que corra eficientemente sobre cierta clase de computadores. En particular se considera si es útil agrupar tareas y si vale la pena replicar datos/o cómputos.

En la fase de partición se trató de establecer el mayor número posible de taras con la intención de explorar al máximo las posibilidades de paralelismo. Esto no necesariamente produce un algoritmo eficiente ya que el costo de comunicación puede ser significativo. En la mayoría de los computadores paralelos la comunicación es mediante el paso de mensajes y frecuentemente ha y que parar los cómputos para enviar o recibir mensajes. Mediante la agrupación de tareas se puede reducir la cantidad de datos a enviar y así reducir el número de mensajes y el costo de comunicación.

La comunicación no sólo depende de la cantidad de información enviada. Cada comunicación tiene un coste fijo de arranque. Reduciendo el número de mensajes, a pesar de que se envíe la misma cantidad de información, puede ser útil. Así mismo, se puede intentar replicar cómputos y/o datos para reducir los requerimientos de comunicación. Por otro lado, también se debe considerar el costo de creación de tareas y el costo de cambio de contexto[12] (*context switch*) en caso de que se asignen varias tareas a un mismo procesador.

En caso de tener distintas tareas corriendo en diferentes computadores con memorias privadas, se debe trata de que la granularidad sea gruesa: exista una cantidad de cómputo significativa antes de tener necesidades de comunicación. Esto se aplica a máquinas como la SP2 y redes de estaciones de trabajo. Se puede tener granularidad media si la aplicación correrá sobre una máquina de memoria compartida. En estas máquinas el costo de comunicación es menor que en las anteriores siempre y cuando en número de tareas y procesadores se mantenga dentro de cierto rango. A medida que la granularidad decrece y el número de procesadores se incrementa, se intensifican las necesidades de altas velocidades de comunicaciones entre los nodos. Esto hace que los sistemas de grano fino por lo general requieran máquinas de propósito específico. Se pueden usar máquinas masivamente paralelas conectadas mediante una red de alta velocidad (arreglos de procesadores o máquinas vectoriales).

2.9.4 Asignación

En esta última etapa se determina en qué procesador se ejecutará cada tarea. Este problema no se presenta en máquinas de memoria compartida tipo UMA. Estas proveen asignación dinámica de procesos y los procesos que necesitan de un CPU están en una cola de procesos listos. Cada procesador tiene acceso a esta cola y puede correr el próximo proceso. No consideraremos más este caso.

Actualmente, no hay mecanismos generales de asignación de tareas para máquinas distribuidas. Esto continúa siendo un problema difícil y que debe ser atacado explícitamente a la hora de diseñar algoritmos paralelos.

La asignación de tareas puede ser estática o dinámica. En la **asignación estática**, las tareas son asignadas a un procesador al comienzo de la ejecución del algoritmo paralelo y corren ahí hasta el final. La asignación estática en

ciertos casos puede reducir el costo de creación de procesos, sincronización y terminación.

En la **asignación dinámica** se hacen cambios en la distribución de las tareas entre los procesadores a tiempo de ejecución, o sea, hay **migración** de tareas a tiempo de ejecución. Esto es con el fin de balancear la carga del sistema y reducir el tiempo de ejecución. Sin embargo, el costo de balanceo puede ser significativo y por ende incrementar el tiempo de ejecución.

Fuentes del paralelismo

El procesamiento paralelo tiene como principal objetivo explotar el paralelismo inherente a las aplicaciones informáticas. Todas las aplicaciones no presentan el mismo perfil cara al paralelismo: unas se pueden paralelizar mucho y en cambio otras muy poco. Al lado de este factor cuantitativo evidente, es necesario considerar también un factor cualitativo: la manera a través de la cual se explota el paralelismo. Cada técnica de explotación del paralelismo se denomina fuente. Distinguimos tres fuentes principales:

Paralelismo de control

La explotación del paralelismo de control proviene de la constatación natural de que en una aplicación existen acciones que podemos “hacer al mismo tiempo”. Las acciones, llamadas también tareas o procesos pueden ejecutarse de manera más o menos independientemente sobre unos recursos de cálculo llamados también procesadores elementales o (PE).

En el caso de que todas las acciones sean independientes es suficiente asociar un recurso de cálculo a cada una de ellas para obtener una ganancia en tiempo de ejecución que será lineal: *N* acciones independientes se ejecutarán *N* veces más rápido sobre *N* elementos de proceso (PE) que sobre un solo. Este es el caso ideal, pero las acciones de un programa real suelen presentar dependencias entre ellas. Distinguiremos dos clases de dependencias que suponen una sobrecarga de trabajo:

- Dependencias de control de secuencia: corresponde a la secuenciación en un algoritmo clásico.
- Dependencia de control de comunicación: una acción envía informaciones a otra acción.

La explotación del paralelismo de control consiste en administrar las dependencias entre las acciones de un programa para obtener así una asignación de recursos de cálculo lo más eficaz posible, minimizando estas dependencias. La figura 17 muestra un ejemplo de paralelismo de control aplicado a la ejecución simultánea de instrucciones.

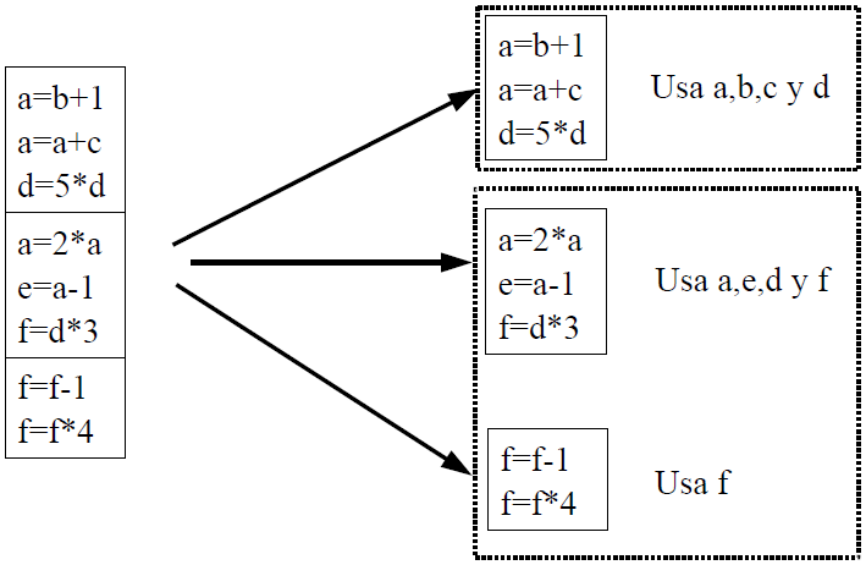

Figura 17 - Ejemplo de paralelismo de control

El paralelismo de datos

La explotación del paralelismo de datos proviene de la constatación natural de que ciertas aplicaciones trabajan con estructuras de datos muy regulares (vectores, matrices) repitiendo una misma acción sobre cada elemento de la estructura. Los recursos de cálculo se asocian entonces a los datos. A menudo existe un gran número (millares o incluso millones) de datos idénticos. Si el número de PE es inferior al de datos, éstos se reparten en los PE disponibles. La figura 18 muestra de forma gráfica el concepto de paralelismo de datos.

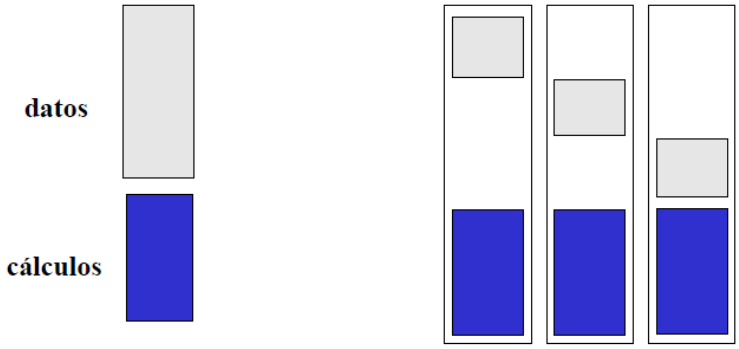

Figura 18 - Paralelismo de datos

Como las acciones efectuadas en paralelo sobre los PE son idénticas, es posible centralizar el control. Siendo los datos similares, la acción a repetir tomará el mismo tiempo sobre todos los PE y el controlador podrá enviar, de manera síncrona, la acción a ejecutar a todos los PE.

Las limitaciones de este tipo de paralelismo vienen dadas por las necesidades de dividir los datos vectoriales para adecuarlos al tamaño soportado por la

máquina, la existencia de datos escalares que limitan el rendimiento y la existencia de operaciones de difusión (un escalar se reproduce varias veces convirtiéndose en un vector) y *β*-reducciones que no son puramente paralelas.

Paralelismo de flujo

La explotación del paralelismo de flujo proviene de la constatación natural de que ciertas aplicaciones funcionan en modo cadena: disponemos de un flujo de datos, generalmente semejantes, sobre los que debemos efectuar una sucesión de operaciones en cascada. La figura 19 muestra de forma gráfica el concepto de paralelismo de flujo.

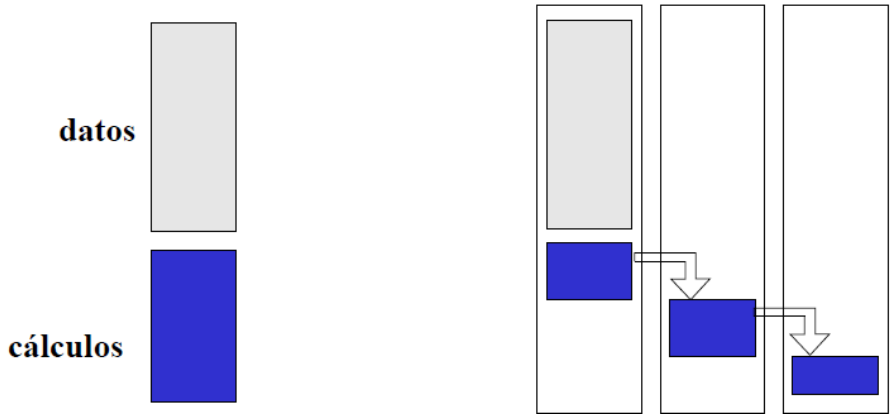

Figura 19 - Paralelismo de flujo

Los recursos de cálculo se asocian a las acciones y en cadena, de manera que los resultados de las acciones efectuadas en el instante *t* pasen en el instante *t+1* al PE siguiente. Este modo de funcionamiento se llama también *segmentación* o *pipeline*.

El flujo de datos puede provenir de dos fuentes:

- Datos de tipo vectorial ubicados en memoria. Existe entonces una dualidad fuerte con el caso del paralelismo de datos.
- Datos de tipo escalar provenientes de un dispositivo de entrada. Este dispositivo se asocia a menudo a otro de captura de datos, colocado en un entorno de tiempo real.

En ambos casos, la ganancia obtenida está en relación con el número de etapas (número de PE). Todos los PEs no estarán ocupados mientras el primer dato no haya recorrido todo el cauce, lo mismo ocurrirá al final del flujo. Si el flujo presenta frecuentes discontinuidades, las fases transitorias del principio y del fin pueden degradar seriamente la ganancia. La existencia de bifurcaciones también limita la ganancia obtenida.

2.10 MPI (*Message Passing Interface*)

2.10.1 Introducción a MPI

MPI (*Message Passing Interface*) es un interfaz estandarizado para la realización de aplicaciones paralelas basadas en paso de mensajes. El modelo de programación que subyace tras MPI es MIMD (*Multiple Instruction streams, Multiple Data streams)* aunque se dan especiales facilidades para la utilización del modelo SPMD (*Single Program Multiple Data*), um caso particular de MIMD em el que todos los procesos ejecutan el mismo programa, aunque no necesariamente la misma instrucción al mismo tiempo[25].

MPI es, como su nombre indica, un interfaz, lo que quiere decir que el estándar no exige una determinada implementación del mismo. Lo importante es dar al programador una colección de funciones para que éste diseñe su aplicación, sin que tenga necesariamente que conocer el hardware concreto sobre el que se va a ejecutar, ni la forma en la que se han implementado las funciones que emplea.

MPI ha sido desarrollado por el MPI Forum, un grupo formado por investigadores de universidades, laboratorios y empresas involucradas em la computación de altas prestaciones. Los objetivos fundamentales del MPI Forum son los siguientes:

1. Definir un entorno de programación único que garantice la portabilidad de las aplicaciones paralelas.

2. Definir totalmente el interfaz de programación, sin especificar cómo debe ser la implementación del mismo.

3. Ofrecer implementaciones de calidad, de dominio público, para favorecer la extensión del estándar.

4. Convencer a los fabricantes de computadores paralelos que ofrezcan versiones de MPI optimizadas para sus máquinas (lo que ya han hecho fabricantes como IBM y Silicon Graphics)

Los elementos básicos de MPI son una definición de un interfaz de programación independiente de lenguajes, más una colección de *bindings* o concreciones de ese interfaz para los lenguajes de programación más extendidos en la comunidad usuaria de computadores paralelos: C y FORTRAN.

Como ya se ha comentado, MPI está especialmente diseñado para desarrollar aplicaciones SPMD. Al arrancar una aplicación se lanzan en paralelo N copias del mismo programa (procesos). Estos procesos no avanzan sincronizados instrucción a instrucción sino que la sincronización, cuando sea necesaria, tiene que ser explícita. Los procesos tienen un espacio de memoria

completamente separado. El intercambio de información, así como la sincronización, se hacen mediante paso de mensajes.

Se dispone de funciones de comunicación punto a punto (que involucran sólo a dos procesos), y de funciones u operaciones colectivas (que involucran a múltiples procesos). Los procesos pueden agruparse y formar comunicadores, lo que permite una definición del ámbito de las operaciones colectivas, así como un diseño modular.

La estructura típica de un programa MPI, usando el *binding* para C, es la siguiente:

```
#include “mpi.h”

main (int argc, char **argv){
        int nproc;  //Número de procesos
        int yo;         //Mi número de proceso

        MPI_Init (&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &nproc);
        MPI_Comm_rank(MPI_COMM_WORLD, &yo);

        /* CUERPO DEL PROGRAMA */
        MPI_Finalize();
}
```

Este segmento de código ya nos presenta cuatro de las funciones más utilizadas de MPI:

*int MPI_Init (int *argc, char **argv)*: Función empleada para iniciar la aplicación paralela.

int MPI_Comm_size (MPI_Comm comm, int *size): Función para averiguar el número de procesos que participan en la aplicación.

int MPI_Comm_rank (MPI_Comm comm, int *rank): Función para que cada proceso averigue su identificador dentro de la colección de procesos que componen la aplicación.

int MPI_Finalize(void): Función para dar por finalizada la aplicación.

2.10.2 Comunicación punto a punto

Un buen número de funciones de MPI están dedicadas a la comunicación entre pares de procesos. Existen múltiples formas distintas de intercambiar un mensaje entre dos procesos[26], en función del modelo y el modo de comunicación elegido.

Modelos y modos de comunicación

MPI define dos modelos de comunicación: bloqueante (*blocking*) y no bloqueante (*nonblocking*). El modelo de comunicación tiene que ver con el tiempo que un proceso pasa bloqueado tras llamar a una función de comunicación, sea ésta de envío o de recepción. Una función bloqueante mantiene a un proceso bloqueado hasta que la operación solicitada finalice. Una no bloqueante supone simplemente “encargar” al sistema la realización de una operación, recuperando el control inmediatamente. El proceso tiene que preocuparse, más adelante, de averiguar si la operación ha finalizado o no.

Queda una duda sin embargo: ¿cuándo queda una operación por finalizada? En el caso de la recepción está claro: cuando tengamos un mensaje nuevo, completo, en el buffer asignado al efecto. En el caso de la emisión la cosa es más compleja: se puede entender que la emisión ha terminado cuando el mensaje ha sido recibido en destino, o se puede ser menos restrictivo y dar por terminada la operación en cuanto se ha hecho una copa del mensaje en un buffer del sistema en el lado emisor. MPI define un envío como finalizado cuando el emisor puede utilizar, sin problemas de causar interferencias, el buffer de emisión que tenía el mensaje. Dicho esto, podemos entender que tras hacer un *send* (envío) bloqueante podemos reutilizar el buffer asociado sin problemas, pero tras hacer un *send* no bloqueante tenemos que ser muy cuidadosos con las manipulaciones que se realizan sobre el buffer, bajo el riesgo de alterar inadvertidamente la información que se está enviando.

Al margen de si la función invocada es bloqueante o no, el programador puede tener un cierto control sobre la forma en la que se realiza y completa un envío. MPI define, en relación a este aspecto, 4 modos de envío: básico (*basic*), con buffer (*buffered*), síncrono (*synchronous*) y listo (*ready*).

Cuando se hace un envío con buffer se guarda inmediatamente, en un buffer el efecto en el emisor, una copia del mensaje. La operación se da por completa en cuanto se ha efectuado esta copia. Si no hay espacio en el buffer, el envío fracasa.

Si se hace un envío síncrono, la operación se da por terminada sólo cuando el mensaje ha sido recibido en destino. Este es el modo de comunicación habitual en los sistemas basados en *Transputers*. En función de la implementación elegida, puede exigir menos copias de la información conforme ésta circula el buffer del emisor al buffer del receptor.

El modo de envío básico no especifica la forma en la que se completa la operación: es algo dependiente de la implementación. Normalmente, equivale a

un envío con buffer para mensajes cortos y a un envío síncrono para mensajes largos. Se intenta así agilizar el envío de mensajes cortos a la vez que se procura no perder demasiado tiempo realizando copias de la información.

En cuanto al envío en modo listo, sólo se puede hacer si antes el otro extremo está preparado para una recepción inmediata. No hay copias adicionales del mensaje (como en el caso del modo con buffer), y tampoco podemos confiar en bloquearnos hasta que el receptor esté preparado.

Comunicación básica

El resultado de la combinación de dos modelos y cuatros modos de comunicación nos da 8 diferentes funciones de envío. Funciones de recepción sólo hay dos, una por modelo. Se presenta a continuación los prototipos de las funciones más habituales.

int MPI_Send (void buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm.);*

int MPI_Recv (void buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm., MPI_Status *status);*

El significado de los parámetros es como sigue. *Buf*, *count* y *datatype* forman el mensaje a enviar o recibir: *count* copias de un dato del tipo *datatype* que se encuentran en memoria a partir de la dirección indicada por *buf*. *Dest* es, en las funciones de envío, el identificador del proceso destinatario del mensaje. *Source* es, en las funciones de recepción, el identificador del emisor del cual esperamos un mensaje. *Tag* es una etiqueta que se puede poner al mensaje. El significado de la etiqueta lo asigna el programador. Suele emplearse para distinguir entre diferentes clases de mensajes. El emisor pone siempre una etiqueta a los mensajes, y el receptor puede elegir entre recibir sólo los mensajes que tengan una etiqueta dada, o aceptar cualquier etiqueta, poniendo *MPI_ANY_TAG* como valor de este parámetro. *Comm* es un comunicador. Status es un resultado que se obtiene cada vez que se completa una recepción, y nos informa de aspectos tales como el tamaño del mensaje recibido, la etiqueta del mensaje y el emisor del mismo. La definición de la estructura *MPI_Status* es la siguiente:

```
typedef struct{

    int MPI_SOURCE;
    int MPI_TAG ;

/* otros campos no accesibles*/
}MPI_Status;
```

Para poder conocer el tamaño del mensaje, se puede emplear la función *MPI_Get_count()*:

*int MPI_Get_count (MPI_Status *status, MPI_Datatype datatype, int *count);*

MPI_Isend() y *MPI_Irecv()* son las funciones emisión/recepción básicas no bloqueantes.

int MPI_Isend (void buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request);*

int MPI_Irecv (void buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request);*

*int MPI_Wait (MPI_Request *request, MPI_Status *status);*
*int MPI_Test (MPI_Request *request, int *flag, MPI_Status *status);*
*int MPI_Cancel (MPI_Request *request);*

Las funciones no bloqueantes manejan un objeto *request* del tipo *MPI_Request*. Este objeto es una especie de "recibo" de la operación solicitada. Más adelante se podrá utilizar este recibo para saber si la operación ha terminado o no.

La función *MPI_Wait*() toma como entrada un recibo, y bloquea al proceso hasta que la operación correspondiente termina. Por lo tanto, hacer un *MPI_Isend*() seguido de un *MPI_Wait*() equivale a hacer un envío bloqueante. Sin embargo, entre la llamada a la función de envío y la llamada a la función de espera el proceso puede haber estado haciendo cosas útiles, es decir, consigue solapar parte de cálculo de la aplicación con la comunicación.

Cuando no interesa bloquearse, sino simplemente saber si la operación ha terminado o no, podemos usar *MPI_Test*(). Esta función actualiza un flag que se le pasa como segundo parámetro. Si la función ha terminado, este flag toma el valor 1, y si no ha terminado pasa a valer 0.

Por último, *MPI_Cancel*() nos sirve para cancelar una operación de comunicación pendiente, siempre que ésta aún no se haya completado.

Comunicación con buffer

Uno de los problemas que tiene el envío básico es que el programador no tiene control sobre cuánto tiempo va a tardar en completarse la operación. Puede que apenas tarde, si es que el sistema se limita a hacer una copia del mensaje en un buffer, que saldrá más tarde hacia su destino; pero también puede que mantenga el proceso bloqueado un largo tiempo, esperando a que el receptor acepte el mensaje.

Para evitar el riesgo de un bloqueo no deseado se puede solicitar explícitamente que la comunicación se complete copiando el mensaje en un buffer, que tendrá que asignar al efecto el propio proceso. Así, se elimina el riesgo de bloqueo mientras se espera a que el interlocutor esté preparado. La función correspondiente es *MPI_Bsend*(), que tiene los mismos argumentos que *MPI_Send*().

int MPI_Bsend (void buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm*);

Para que se pueda usar el envío con buffer es necesario que el programador asigne un buffer de salida. Para ello hay que reservar previamente una zona de memoria (de forma estática o con un *malloc*()) y luego indicarle al sistema que la emplee como buffer. Esta última operación la hace *MPI_Buffer_attach*(). Ese buffer se puede recuperar usando *MPI_Buffer_detach()*.

int MPI_Buffer_attach (void buffer, int size);*
int MPI_Buffer_detach (void buffer, int* size);*

Recordemos que una peculiaridad de los envíos con buffer es que fracasan en el caso de que el buffer no tenga suficiente espacio como para contener un mensaje, y el resultado es que el programa aborta. El programador puede decidir entonces que el buffer asignado es muy pequeño, y asignar uno más grande para una ejecución posterior, o bien caminar el modo de comunicación a otro con menos requerimientos de buffer pero con más riesgo de bloqueo.

Recepción por encuesta

Las funciones de recepción de mensajes engloban en una operación la sincronización con el emisor (esperar a que haya un mensaje disponible) con la de comunicación (copiar este mensaje). A veces, sin embargo, conviene separar ambos conceptos. Por ejemplo, podemos estar a la espera de mensajes de tres clases, cada una asociada a un tipo de datos diferente, y la clase nos viene dada por el valor de la etiqueta. Por lo tanto, nos gustaría saber el valor de la etiqueta antes de leer el mensaje. También puede ocurrir que nos llegue un mensaje de longitud desconocida, y resulte necesario averiguar primero el tamaño para así asignar dinámicamente el espacio de memoria requerido por el mensaje.

Las funciones *MPI_Probe*() y *MPI_Iprobe*() nos permiten saber si tenemos un mensaje recibido y listo para leer, pero sin leerlo. A partir de la información de estado obtenida con cualquier de estas “sondas”, podemos averiguar la identidad del emisor del mensaje, la etiqueta del mismo y su longitud. Una vez hecho esto, podemos proceder a la lectura real del mensaje con la correspondiente llamada a *MPI_Recv*() o *MPI_Irecv*().

*int MPI_Iprobe (int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status)*
*int MPI_Probe (int source, int tag, MPI_Comm comm, MPI_Status *status)*

MPI_Probe() es bloqueante: sólo devuelve el control al proceso cuando hay un mensaje listo. *MPI_Iprobe*() es no bloqueante: nos indica en el argumento *flag* si hay un mensaje listo o no, es decir, realiza una encuesta.

2.10.3 Operaciones colectivas

Muchas aplicaciones requieren la comunicación entre más de dos procesos. Esto se puede hacer combinando operaciones punto a punto, pero para que le resulte más cómodo al programador, y para posibilitar implementaciones optimizadas, MPI incluye una serie de operaciones colectivas:

- Barreras de sincronización.
- *Broadcast* (difusión)
- *Gather* (recolección)
- *Scatter* (distribución)
- Operaciones de reducción (suma, multiplicación, mínimo, etc)
- Combinaciones de todas ellas

Una operación colectiva tiene que ser invocada por todos los participantes, aunque los roles que jueguen no sean los mismos. La mayor parte de las operaciones requieren la designación de un proceso como *root*, o raíz de la operación.

Barreras y broadcast

Dos de las operaciones colectivas más comunes son las barreras de sincronización (*MPI_Barrier()*) y el envío de información en modo difusión (*MPI_Broadcast()*). La primera de estas operaciones no exige ninguna clase de intercambio de información. Es una operación puramente de sincronización, que bloquea a los procesos de un comunicador hasta que todos ellos han pasado por la barrera. Suele emplearse para dar por finalizada una etapa del programa, asegurándose de que todos han terminado antes de dar comienzo a la siguiente.

int MPI_Barrier (MPI_Comm comm);

MPI_Broadcast() sirve para que un proceso, el raíz, envíe un mensaje a todos los miembros del comunicador. Esta función ya muestra una característica peculiar de las operaciones colectivas de MPI: todos los participantes invocan la misma función, designando al mismo proceso como raíz de la misma. Una implementación alternativa sería tener una función de envío especial, en modo *broadcast*, y usar las funciones de recepción normales para leer los mensajes.

int MPI_Bcast (void buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm.);*

Este esquema representa el significado de un *broadcast*. En las filas se representan los procesos de un comunicador, y en las columnas los datos que posee cada proceso. Antes del *broadcast*, el procesador raíz (suponemos que es el 0) tiene el dato A0. Tras realizar el *broadcast*, todos los procesos (incluyendo el raíz) tienen una copia de A0.

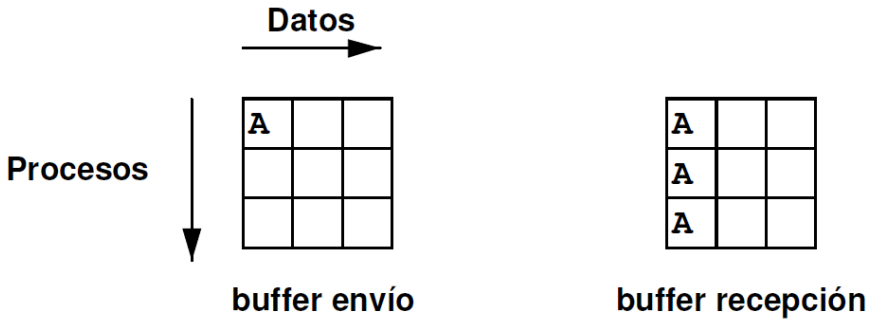

Figura 20 - Esquema de la operación colectiva MPI_Broadcast(). "Buffer envío" indica los contenidos de los bufferes de envío antes de proceder a la operación colectiva. "Buffer recepción" indica los contenidos de los bufferes de recepción tras completarse la operación

Recolección (*gather*)

MPI_Gather realiza una recolección de datos en el proceso raíz. Este proceso recopila un vector de datos, al que contribuyen todos los procesos del comunicador con la misma cantidad de datos. El proceso raíz almacena las contribuciones de forma consecutiva.

int MPI_Gather (void sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvbuf, MPI_Datatype recvtype, int root, MPI_Comm comm);*

Figura 21 - Esquema de la operación colectiva MPI_Gather()

Sólo el proceso raíz tiene que preocuparse de los parámetros *recbuf*, *recvcount* y *recvtype*. Sin embargo, todos los procesos (raíz incluida) tienen que facilitar valores válidos para *sendbuf*, *sendcount* y *sendtype*.

Existe una versión de la función de recolección, llamada *MPI_Gatherv*(), que permite almacenar los datos recogidos en forma no consecutiva, y que cada proceso contribuya con bloques de datos de diferente tamaño. La tablas *recvcounts* establece el tamaño del bloque de datos aportado por cada proceso, y *dipsis* indica cuál es el desplazamiento entre bloque y bloque.

int MPI_Gatherv (void sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, int root, MPI_Comm comm);*

Muchas veces interesa distribuir a todos los procesos el resultado de una recolección previa. Esto se puede hacer concatenando una recolección con un *broadcast*. Existen funciones que se encargan de hacer todo esto en un único paso: *MPI_Allgather*() (si los bloques de datos son de tamaño fijo y se almacenan sucesivamente) y *MPI_Allgatherv*() (si los tamaños de los datos son variables y/o se almacenan de forma no consecutiva).

int MPI_Allgather (void sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm);*

int MPI_Allgatherv (void sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, MPI_Comm comm);*

Figura 22 - Esquema de la operación colectiva MPI_Allgather()

Distribución (*scatter*)

MPI_Scatter() realiza la operación simétrica a *MPI_Gather*(). El proceso raíz posee un vector de elementos, uno por cada proceso del comunicador. Tras realizar la distribución, cada proceso tiene una copia del vector inicial. Recordar que MPI permite enviar bloques de datos, no sólo datos individuales.

int MPI_Scatter (void sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);*

Si los datos a enviar no están almacenados de forma consecutiva en memoria, o los bloques a enviar a cada proceso no son todos del mismo tamaño, tenemos la función *MPI_Scatterv*(), con una interfaz más complejo, pero más flexible.

int MPI_Scatterv (void sendbuf, int *sendcounts, int *displs, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);*

Figura 23 - Esquema de la operación colectiva MPI_Scatter()

Comunicación todos con todos (All-to-all)

La comunicación de todos con todos supone que, inicialmente, cada proceso tiene un vector con tantos elementos como procesos hay en el comunicador. Para *i,j* y *k* entre 0 y *N-1* (donde *N* es el número de procesos del comunicador), cada proceso i envía una copia de *sendbuf[j]* al proceso j, y recibe del proceso k un elemento, que almacena en *recvbuf[k]. MPI_Alltoall()* equivale, por tanto, a una sucesión de N operaciones de distribución, en cada un de a las cuales el proceso i toma el rol de raíz.

int MPI_Alltoall (void sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm);*

int MPI_Alltoallv (void sendbuf, int *sendcounts, int *sdispls, MPI_Datatype sendtype, void* recvbuf, int *recvcounts, int *rdispls, MPI_Datatype recvtype, MPI_Comm comm);*

Figura 24 - Esquema de la operación colectiva MPI_Alltoall()

Reducción

Una reducción es una operación realizada de forma cooperativa entre todos los procesos de un comunicador, de tal forma que se obtiene un resultado final que se almacena en el proceso raíz.

int MPI_Reduce (void sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);*

MPI define una colección de operaciones del tipo *MPI_Op* que se pueden utilizar en una reducción: *MPI_MAX (máximo), MPI_MIN (mínimo), MPI_SUM (suma), MPI_PROD (producto), MPI_LAND (and lógico), MPI_BAND (and bit a bit), MPI_LOR (or lógico), MPI_BOR (or bit a bit), MPI_LXOR (xor lógico), MPI_BXOR (xor bit a bit)*, etc.

En ocasiones el programador puede necesitar otra operación de reducción distinta, no predefinida. Para ello MPI ofrece la función *MPI_Op_create*(), que toma como argumento de entrada una función de usuario y devuelve un objeto de tipo *MPI_Op* que se puede emplear con *MPI_Reduce*(). Ese objeto se puede destruir más tarde con *MPI_Op_free*().

*int MPI_Op_create (MPI_User_function *function, int conmute, MPI_Op op);*
*int MPI_op_free (MPI_Op *op);*

La operación a realizar puede ser cualquiera. En general se emplean operaciones conmutativas y asociativas, es decir, cuyo resultado no depende del orden con el que se procesan los operandos. Eso se indica con el valor 1 en el parámetro conmute. Si la operación no es conmutativa (*conmute = 0*) entonces se exige que la reducción se realice en orden de dirección (se empieza con el proceso 0, luego con el 1, con el 2, etc.).

En ocasiones resulta útil que el resultado de una reducción esté disponible para todos los procesos. Aunque se puede realizar un *broadcast* tras la reducción, podemos combinar la reducción con la difusión usando *MPI_Allreduce*(). Si el resultado de la reducción es un vector que hay que distribuir entre los procesadores, podemos combinar la reducción y la distribución usando *MPI_Reduce_scatter*().

int MPI_Allreduce (void sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);*

int MPI_Reduce_scatter (void sendbuf, void* recvbuf, int *recvcounts, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);*

Una variante más de la reducción nos da la *MPI_Scan*(). Es similiar a *MPI_Allreduce*(), excepto que cada proceso recibe un resultado parcial de la reducción, en vez del final: cada proceso *i* recibe, en vez del resultado de OP(0,…,N-1), siendo *N* el número total de procesos- el resultado de *OP(0,…,i).*

int MPI_Scan (void sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);*

2.10.4 Tipos de datos derivados

MPI maneja en todas sus funciones de envío/recepción vectores de tipo simples. En general, se asume que los elementos de esos vectores están almacenados consecutivamente en memoria. En ocasiones, sin embargo, es necesario el intercambio de tipos estructurados (*structs*) [27][28], o de vectores no almacenados consecutivamente en memoria (por ejemplo, envío de una columna de un array, en vez de envío de una fila).

MPI incluye la posibilidad de definir tipos más complejos (objetos del tipo *MPI_Datatype*), empleando constructores. Antes de usar un tipo de usuario, hay que ejecutar *MPI_Type_commit*(). Cuando ya no se necesite un tipo, se puede liberar con *MPI_Type_free*().

*int MPI_Type_commit (MPI_Datatype *datatype);*
*int MPI_Type_free (MPI_Datatype *datatype);*

Definición de tipos homogéneos

Son tipos homogéneos aquellos tipos construidos en los que todos los elementos constituyentes son del mismo tipo. Se pueden definir tipos homogéneos con dos funciones distintas: *MPI_Type_contiguous*() y *MPI_Type_vector*(). La primera versión es la más sencilla, y permite definir un tipo formado por una colección de elementos de un tipo básico, todos ellos del mismo tamaño y almacenados consecutivamente en memoria.

*int MPI_Type_contiguous (int count,. MPI_Datatype oldtype, MPI_Datatype *newtype);*

Newtype es un nuevo tipo que consiste en *count* copias de *oldtype*. Si los elementos constituyentes del nuevo tipo no están almacenados consecutivamente en memoria, sino que están espaciados a intervalos regulares, la función a emplear es *MPI_Type_vector*().

*int MPI_Type_vector (int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype);*

Newtype es un nuevo tipo que consiste en *count* bloques de datos. Cada bloque consta de *blocklength* elementos del tipo *oldtype*. La distancia entre bloques, medida en múltiplos del tamaño del elemento básico, la da *stride*.

Una llamada a *MPI_Type_contiguous(c,o,n)* equivale a una llamada a *MPI_Type_vector(c,1,1,o,n),* o a una llamada a *MPI_Type_vector(1,c,x,o,n),* siendo x un valor arbitrario.

Definición de tipos heterogéneos

Si nos interesa trabajar con tipos no homogéneos, podemos usar una función más genérica: *MPI_Type_struct().*

*MPI_Type_struct (int count, int *array_of_blocklenght, MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types, MPI_Datatype *newtype);*

Count determina el número de bloques, así como el tamaño de los arrays *array_of_blocklenghts*, *array_of_displacements* y *array_of_types*. Cada bloque i tiene *array_of_blocklenghts*[i] elementos, está desplazado *array_of_displacements*[i] bytes del anterior, y es de tupo *array_of_types*[i].

2.11 OpenMP

OpenMP es una interfaz de programación de aplicaciones (API) para la programación de memoria compartida en múltiples plataformas. Permite añadir concurrencia a los programas escritos en C, C++ y Fortran sobre la base del modelo de ejecución fork-join. Está disponible en muchas arquitecturas, incluidas las plataformas Unix y Microsoft Windows. Se compone de un conjunto de directivas de compilador, rutinas de biblioteca y variables de entorno que influencian el comportamiento en tiempo de ejecución[29].

- El **conjunto de directivas de compilador** usado por el programador empleadas para comunicarse con el compilador en paralelismo.

- La **librería de funciones en tiempo de ejecución** que habilita la colocación e interroga sobre los parámetros paralelos que se van a usar, tal como número de los hilos que van a participar y el número de cada hilo.

- Un número limitado de **variables de entorno** que pueden ser usadas para definir en tiempo de ejecución parámetros del sistema en paralelo tales como el número de hilos.

La especificación OpenMP es diseñada por los miembros del OpenMP Architecture Review Board (OpenMP ARB) los cuales se encargan de mantener y revisar la especificación OpenMP. Los miembros actuales del OpenMP ARB son:

Miembros Permanentes del ARB:

- AMD (Roy Ju)
- Cray (James Beyer)
- Fujitsu (Matthijs van Waveren)
- HP (Uriel Schafer)
- IBM (Kelvin Li)
- Intel (Sanjiv Shah)
- NEC (Kazuhiro Kusano)
- The Portland Group, Inc. (Michael Wolfe)
- Oracle Corporation (Nawal Copty)
- Microsoft (-)
- Texas Instruments (Andy Fritsch)
- CAPS-Entreprise (Francois Bodin)

Miembros auxiliaries de ARB:

- ANL (Kalyan Kumaran)
- ASC/LLNL (Bronis R. de Supinski)
- cOMPunity (Barbara Chapman)

- EPCC (Mark Bull)
- LANL (John Thorp)
- NASA (Henry Jin)
- RWTH Aachen University (Dieter an Mey)

Historia de las especificaciones OpenMP:

- 1997: OpenMP 1.0 Fortran
- 1998: OpenMP 1.0 C/C++
- 1999: OpenMP 1.1 Fortran
- 2000: OpenMP 2.0 Fortran
- 2002: OpenMP 2.0 C/C++
- 2005: OpenMP 2.5 Fortran, C/C++
- 2008: OpenMP 3.0 Fortran, C/C++

OpenMP utiliza el modelo de ejecución paralela fork-join. El programa comienza su ejecución con un único hilo, llamado maestro, que cuando se encuentra la primera construcción paralela crea un conjunto de hilos que junto con el maestro se reparten el trabajo de la ejecución. Una vez acabada la región paralela sólo el maestro continúa y el resto de hilos se destruyen.

Este lenguaje de directivas generalmente se emplea para la paralelización de lazos. El procedimiento es buscar los lazos computacionalmente más costosos y que los hilos se repartan las iteraciones del lazo. Durante la ejecución de la región paralela los hilos se comunican a través de variables compartidas. Es necesario tener cuidado al programar ya que la compartición de datos puede llevar a un mal comportamiento del programa debido al acceso simultáneo a los mismo desde diferentes nodos. El modo de evitarlo es usando directivas de sincronización que tienen la desventaja de ser computacionalmente costosas, por lo que vamos a trata de evitarlas.

Características principales del estándar OpenMP

OpenMP dispone de una sintaxis general de las directivas de forma centinela nombre_directiva [cláusulas]. Como centinelas se emplean C$OMP ó *$OMP en programas de formato fijo y ¡$omp en programas de formato fijo o variable. Las cláusulas pueden estar separadas por espacios o comas y su orden es indiferente. Además, permite la compilación condicional de líneas de programa por medio de centinelas, por ejemplo si introducimos la línea de código en lenguaje Fortran,

```
¡$ A(i)=A(i)+b
```

cuando compilamos sin +Oopenmp el programa no ejecuta esta línea. El mismo ejemplo para lenguaje C y C++ sería de la forma,

```
#ifdef _OPENMP
A(i)=A(i)+b
#endif
```

De modo que esta línea solamente forma parte del programa cuando se compila con OpenMP activado[30].

La programación con OpenMP consta básicamente de tres elementos, el control de paralelismo, el control de datos y comunicaciones y finalmente el control de la sincronización. Para el control de paralelismo existen directivas para la creación de regiones paralelas y para el reparto de trabajo. El control de datos y comunicaciones se realiza mediante el uso de variables privadas y compartidas. Finalmente la sincronización nos permite coordinar el acceso a los datos mediante el uso de barreras, secciones críticas y otras opciones que se verán más adelante.

Directivas para la construcción de paralelismo

Las directivas para el control de paralelismo nos permiten crear regiones paralelas para repartir las tareas entre los hilos que hemos creado. La directiva ***parallel*** es la encargada de crear la región paralela, que es un bloque de código ejecutado en paralelo por varios hilos. Esta directiva indica en OpenMP cuando empieza una región paralela y tiene la siguiente forma:

Fortran	C/C++
¡$OMP PARALLEL [cláusulas] **bloque estructurado** **¡$OMP END PARALLEL**	**#pragma omp nombre_directiva [cláusulas]** **{** **bloque estructurado** **}**

Tabla 2 - Definición directiva de OpenMP *Parallel*

En C/C++ tanto las directivas como las cláusulas se escriben siempre con minúsculas.

Fortran acepta tanto minúsculas como mayúsculas.

Cuando un hilo se encuentra la directiva ***parallel***, crea un equipo de hilos y éste se convierte en el hilo maestro del equipo. El número de hilos que se crean lo controlamos por medio de una variable de entorno ***NUM_THREADS*** o mediante una llamada a la librería.

La directiva ***end parallel*** denota el final de la región paralela. Existe una barrera implícita cuando los hilos llegan a este punto, de modo que el programa no continúa hasta que todos los hilos hayan completado la ejecución del bloque de código, salvo que exista la cláusula ***NOWAIT***. Cuando los hilos alcanzan el final de la región paralela se destruye y tan sólo el hilo maestro continúa la ejecución del programa.

En el caso de que un equipo de hilos se encuentre, durante la ejecución de una región paralela, con otra región paralela, cada hilo crea un nuevo equipo y se convierte en el maestro de ese nuevo equipo. La segunda región paralela se denomina región paralela anidada.

El resto de directivas que se describen a continuación, se emplean para el reparto de trabajo. Estas directivas se encuentran dentro de una región paralela, no crean nuevos hilos y tampoco existen barreras al inicio del reparto de tareas. El primer hilo que llega realiza el trabajo que se le asigna sin esperar a que lleguen los demás.

A continuación, se describe con más detalle cada una de las directivas de reparto de trabajo:

Directiva Do. Esta directiva especifica las iteraciones del primer lazo Do situado justo después de la directiva se deben ejecutar en paralelo. Las iteraciones del lazo se distribuyen entre los hilos ya existentes. Si queremos elegir cómo se reparten los hilos de las iteraciones del lazo podemos hacer uso de la cláusula ***schedule(type,[chunk])***. Existen varias formas de planificar el reparto de iteraciones en función de la sintaxis que hayamos elegido dentro de las que se muestran a continuación:

- ***schedule (static, chunk)***. Cuando especificamos esta opción las iteraciones se dividen en bloques del tamaño especificado en *chunk*. Los bloques se asignan de modo estático, de forma ordenada, entre los hilos del equipo.

- **schedule (dynamic, chunk)**. En esta opción las iteraciones se dividen en bloques de tamaño igual al valor de chunk. Cuando un hilo acaba un bloque de iteraciones dinámicamente se le asigna otro bloque de tamaño igual a chunk.

- **schedule (guided, chunk)**. En esta opción las iteraciones se dividen en bloques de forma que el tamaño de cada bloque decrece de forma exponencial. *Chunk* especifica el tamaño del bloque más pequeño, con excepción del tamaño del último bloque que puede ser menor que *chunk*.

- **schedule (runtime)**. En esta opción el tipo de planificación y el tamaño se puede elegir en tiempo de ejecución por medio de la variable de entorno OMP_SCHEDULE.

Cuando no se especifica la clásula ***schedule(type,[chunk])*** la planificación depende de la implementación.

Los hilos que completan la ejecución del lazo, deben esperar al resto de hilos que todavía no completaron la ejecución, debido a que existe una barrera implícita en la directiva ***END DO***, salvo que exista la clásula ***NOWAIT***.

Directiva SECTIONS. Esta es una directiva de resparto no iterativa, que especifica que la región de código que se encuentra dentro de la directiva se reparte entre los hilos del equipo. Cada sección se ejecuta a su vez por un único hilo del equipo. El formato de la directiva es el siguiente:

Fortran	C/C++

C$OMP SECTIONS [cláusulas]
C$OMP SECTION
bloque estructurado
C$OMP SECTION
bloque estructurado
….
C!$OMP END SECTIONS [NOWAIT]
	#pragma omp sections [clausulas]
	{
	#pragma omp section
	bloque estructurado
	#pragma omp section
	bloque estructurado
	…
	}

Tabla 3 - Definición directiva de OpenMP *Section*

Directiva SINGLE. Con esta directiva se especifica que el código encerrado debe ser ejecutado sólo por un único hilo del equipo. Aquellos hilos que no ejecuten la directiva tienen una barrera implícita al final de la directiva ***SINGLE*** salvo que aparezca la directiva ***NOWAIT***. El formato de esta directa es el que se muestra a continuación:

Fortran	C/C++

C$OMP SINGLE [cláusulas]
bloque estructurado
C!$OMP END SINGLE [NOWAIT]
	#pragma omp single [clausulas]
	{
	bloque estructurado
	}

Tabla 4 - Definición directiva de OpenMP *Single*

También, es posible combinar las directivas para construcción de paralelismo con las directivas para reparto de tareas, para ello tan sólo tenemos que hacer uso de la directiva PARALLEL seguida de la directiva de reparto de tareas que convenga.

Directivas de sincronización

Cuando disponemos de varios hilos, podemos gestionar el modo en el que cada uno accede a determinados bloques de código o los accesos a memoria[31]. Para realizar esta tarea disponemos de directivas que nos permiten gestionar los hilos creados en las regiones paralelas. A continuación, se describe cada una de estas directivas:

MASTER. El bloque de código que se encuentra entre las directivas ***MASTER*** y ***END MASTER*** tan sólo va a ser ejecutado por el hilo maestro, el resto de hilos saltan el bloque de código y continúan con la ejecución. El formato de esta directiva se muestra a continuación:

Fortran	C/C++
¡$OMP MASTER *Bloque estructurado* *¡$OMP END MASTER*	*#pragma omp master* *Bloque estructurado*

Tabla 5 - Definición directiva de OpenMP *Master*

CRITICAL. Esta directiva restringe el acceso a la región de código a un solo hilo. El formato de esta directiva es el que se muestra a continuación:

Fortran	C/C++
¡$OMP CRITICAL [(nombre)] *Bloque estructurado* *¡$OMP END CRITICAL [(nombre)]*	*#pragma omp critical [(nombre)]* *Bloque estructurado*

Tabla 6 - Definición directiva de OpenMP *Critical*

En donde *nombre* identifica a la sección crítica.

Cuando un hilo alcanza esta directiva durante la ejecución, espera al principio de la sección crítica hasta que ningún otro hilo esté ejecutando la sección crítica identificada con el mismo nombre. Las secciones críticas son entidades globales del programa. Si el nombre de una sección crítica coincide con el nombre de otra entidad del programa, el comportamiento del programa es imprevisible.

BARRIER. La función de esta directiva es sincronizar todos los hilos de la región paralela. Durante la ejecución del código cada hilo que encuentra esta directiva espera hasta que todos los hilos alcanzan este punto. El formato que emplea esta directiva es:

Fortran	C/C++
¡$OMP BARRIER	*#pragma omp barrier*

Tabla 7 - Definición directiva de OpenMP *Barrier*

ATOMIC. Con esta directiva nos aseguramos que una posición específica de memoria se actualiza automáticamente de forma atómica (indivisible), lo que implica que no se van a producir múltiples escrituras desde diferentes hilos. Esta directiva se aplica a la sentencia inmediatamente posterior, y debe tener la siguiente forma:

Fortran	C/C++
¡$OMP ATOMIC	*#pragma omp atomic*

Tabla 8 - Definición directiva de OpenMP *Atomic*

FLUSH. Es un punto de encuentro que nos permite asegurar que todos los hilos de un equipo tienen una visión consistente de ciertas variables de memoria. Esta directiva proporciona consistencia entre operaciones de memoria del hilo actual y la memoria global. Para alcanzar consistencia global cada hilo debe ejecutar una operación ***FLUSH***. En el caso de que no especifiquemos una lista de variables se aplica a todas, pero esto podría ser computacionalmente muy costoso. La forma de esta directiva es la que se muestra a continuación:

Fortran	C/C++
¡$OMP FLUSH [(lista variables)]	*#pragma omp flush [(lista variables)]*

Tabla 9 - Definición directiva de OpenMP *Flush*

Esta directiva está presente de modo implícito, salvo que se use la directiva ***NOWAIT***, en las siguientes directivas:

- BARRIER
- CRITICAL Y END CRITICAL
- END DO
- END SECTIONS
- END SINGLE
- ORDERED y END ORDERED
- PARALLEL y END PARALLEL
- PARALLEL DO y END PARALLEL DO
- PARALLEL SECTIONS y END PARALLEL SECTIONS

ORDERED. Las iteraciones del código encerrado entre las directives ORDERED Y END ORDERED se ejecutan en el orden que lo harían en un programa secuencial. Cuando empleamos esta directiva dentro de lazos, la directiva DO o PARALLEL DO tiene que incluir la cláusula ORDERED. La forma de esta directiva es la que se muestra a continuación:

Fortran	C/C++
¡$OMP ORDERED *Bloque estructurado* *¡$OMP END ORDERED*	*#pragma omp ordered* *Bloque estructurado*

Tabla 10 - Definición directiva de OpenMP *Ordered*

Esta directiva secuencializa y ordena el código dentro de la sección ORDERED a la vez que permite ejecutar en paralelo el resto.

Cláusulas para el control de datos

Existe un conjunto de cláusulas, generalmente en las directivas de creación de paralelismo y de reparto de trabajo, que nos permiten mantener un control sobre que hilos comparten, o no comparten, ciertas variables. A continuación, se hablará con más detalle de cada una de estas cláusulas y de la directiva para el control de datos ***THREADPRIVATE***.

THREADPRIVATE. Esta directiva declara las variables que se le indican, privadas a cada hilo pero globales dentro del hilo. Debemos especificar esta directiva cada vez que la variable es declarada. Los datos ***THREADPRIVATE*** serán indefinidos dentro de la región paralela a menos que se use la cláusula ***COPYIN*** en la directiva ***PARALLEL***. La forma de esta directiva es la que se muestra a continuación:

Fortran	C/C++
¡$OMP THREADPRIVATE (lista de variables)	*#pragma omp threadprivate (lista de variables)*

Tabla 11 - Definición directiva de OpenMP *Threadprivate*

Las cláusulas que podemos emplear para el control de datos son las siguientes:

PRIVATE. Esta cláusula declara privadas a cada hilo las variables especificadas. Cada hilo tiene una copia local de las variables, siendo ésta invisible para los demás hilos. Las variables no están definidas ni al entran ni al salir de la región paralela, tan sólo se utilizan dentro de la región paralela. La forma de esta cláusula es la que se muestra a continuación:

PRIVATE (lista de variables)

SHARED. Esta cláusula hace que todas las variables que aparecen en la lista se compartan entre los hilos. El valor asignado a una variable compartida es visto por todos los hilos del equipo, así todos los hilos acceden a la misma posición de memoria cuando modifican variables compartidas y este acceso puede ser concurrente. En el acceso a los datos no se garantiza la exclusión mutua. La forma de esta cláusula es la que se muestra a continuación:

SHARED (lista de variables)

El uso de variables compartidas está recomendando cuando tenemos variables de sólo lectura a las que se accede desde diferentes hilos, los distintos hilos acceden a localizaciones diferentes de la variable (por ejemplo en una matriz cada hilo accede a diferentes índices de la matriz) y cuando queremos comunicar valores entre diferentes hilos.

DEFAULT. Con esta cláusula podemos establecer por defecto que todas las variables de una región paralela sean privadas cuando va seguida de la cláusula ***PRIVATE*** o compartidas cuando va seguida de la cláusula ***SHARED***. En el caso de que no queramos especificar un tipo por defecto podemos hacer uso de la cláusula ***DEFAULT NONE***. De esta forma todas las variables deben clasificarse de forma explícita como privadas o compartidas. La forma de esta cláusula es la siguiente:

DEFAULT (PRIVATE | SHARED | NONE)

Cuando no hacemos uso de la cláusula ***DEFAULT*** el comportamiento por defecto es el mismo que si se hubiese especificado ***DEFAULT(SHARED)***.

FIRSTPRIVATE. Esta cláusula nos permite declarar privadas las variables especificadas y además las inicializa a los valores que tenían antes de entrar en la región paralela. La forma de esta cláusula es la siguiente:

FIRSTPRIVATE (lista de variables)

LASTPRIVATE. Esta cláusula declara privadas las variables de una lista y además actualiza las variables con su último valor al terminar el trabajo paralelo. Cuando una variable se declara como ***LASTPRIVATE***, el valor de la variable al salir del bucle es el valor que toma en el procesador que ejecuta la última iteración del bucle.

REDUCTION. Indica una operación de reducción sobre las variables especificadas. La forma de esta cláusula es la que se muestra a continuación:

REDUCTION (operador:lista de variables)

Una operación de reducción es una aplicación de un operador binario conmutativo y asociativo a una variable y algún otro valor, almacenando el resultado en la variable, como por ejemplo, x=x+a(i).

COPYPRIVATE. Con esta cláusula el valor de las variables privadas son copiadas a las variables privadas de los otros hilos al final de la directiva ***SINGLE***. Por consiguiente esta cláusula sólo puede usarse con esta directiva ***SINGLE***. La forma de esta cláusula es la siguiente:

COPYPRIVATE (lista de variables)

2.12 El rendimiento de los sistemas paralelos

En esta sección se definirán algunas de las medidas más utilizadas a la hora de determinar el rendimiento de una arquitectura paralela. Así, se introducen los conceptos de: *speed-up*, eficiencia de un sistema, utilización, redundancia, etc.

Eficiencia, redundancia, utilización y calidad

Ruby Lee (1980) definió varios parámetros para evaluar el cálculo paralelo. A continuación, se muestra la definición de dichos parámetros[32].

Eficiencia del sistema. Sea *O(n)* el número total de operaciones elementales realizadas por un sistema con *n* elementos de proceso, y *T(n)* el tiempo de ejecución en pasos unitarios de tiempo. En general, T(n) < *O(n)* si los n procesadores realizan más de una operación por unidad de tiempo, donde $n \geq 2$. Supongamos que *T(1) = O(1)* en un sistema mono-procesador. El factor de mejora del rendimiento (*speed-up*) se define como:

$$S(n) = T(1)/T(n)$$

La eficiencia del sistema para un sistema con *n* procesadores se define como:

$$E(n) = \frac{S(n)}{n} = \frac{T(1)}{nT(n)}$$

La eficiencia es una comparación del grado de *speed-up* conseguido frente al valor máximo. Dado que $1 \le S(n) \le n$, tenemos $1/n \le E(n) \le 1$.

La eficacia más baja ($E(n) \to 0$) corresponde al caso en que todo el programa se ejecuta en un único procesador de forma serie. La eficiencia máxima ($E(n) \to 1$) se obtiene cuando todos los procesadores están siendo completamente utilizados durante todo el periodo de ejecución.

Escalabilidad. Un sistema se dice que es escalable para un determinado rango de procesadores *[1,…, n],* si la eficiencia *E(n)* del sistema se mantiene constante y cercana a la unidad en todo ese rango. Normalmente todos los sistemas tienen un determinado número de procesadores a partir del cual la eficiencia empieza a disminuir de forma más o menos brusca. Un sistema es más escalable que otro si este número de procesadores, a partir del cual la eficiencia disminuye, es menor que el otro.

No hay que confundir escalabilidad con ampliabililidad. Un sistema es ampliable si físicamente se le pueden poner más módulos (más memorias, más procesadores, más tarjetas de entrada/salida, etc.). Que un sistema se ampliable no significa que sea escalable, es decir, que un sistema sea capaz de ampliarse con muchos procesadores no significa que el rendimiento vaya a aumentar de forma proporcional, por lo que la eficiencia no tiene por qué mantenerse constante y por tanto el sistema podría no ser escalable.

Redundancia y utilización. La redundancia en un cálculo paralelo se define como la relación entre *O(n)* y *O (1):*

$$R(n) = O(n)/O(1)$$

Esta proporción indica la relación entre el paralelismo software y hardware. Obviamente, $1 \le R(n) \le n$. La utilización del sistema es un cálculo paralelo se define como:

$$U(n) = R(n)E(n) = \frac{O(n)}{nT(n)}$$

La utilización del sistema indica el porcentaje de recursos (procesadores, memoria, recursos, etc.) que se utilizan durante la ejecución de un programa paralelo. Es interesante observar la siguiente relación: $1/n \le E(n) \le U(n) \le 1$ y $1 \le R(n) \le 1/E(n) \le n$.

Calidad del paralelismo. La calidad de un cálculo paralelo es directamente proporcional al *speed-up* y la eficiencia, inversamente proporcional a la redundancia. Así, tenemos:

$$Q(n) = \frac{S(n)E(n)}{R(n)} = \frac{T^3(1)}{nT^2(n)O(n)}$$

Dado que *E(n)* es siempre una fracción y *R(n)* es un número entre 1 y n, la calidad *Q(n)* está siempre limitada por el *speed-up S(n)*.

Para terminar con esta discusión acerca de los índices del rendimiento, usamos el *speed-up S(n)* para indicar el grado de ganancia de velocidad de una computación paralela. La eficiencia *E(n)* mide la porción útil del trabajo total realizado por n procesadores. La redundancia *R(n)* mide el grado del incremento de la carga.

La utilización *U(n)* indica el grado de utilización de recursos durante un cálculo paralelo. Finalmente, la calidad *Q(n)* combina el efecto del *speed-up*, eficiencia y redundancia en una única expresión para indicar el mérito relativo de un cálculo paralelo sobre un sistema.

Perfil del paralelismo en programas

El grado de paralelismo refleja cómo el paralelismo software se adapta al paralelismo hardware. En primer lugar, caracterizamos el perfil del paralelismo de un programa para a continuación introducir los conceptos de paralelismo medio y definir el *speed-up* ideal en una máquina con recursos ilimitados.

Grado de paralelismo[33]. *Es el número de procesos paralelos en los que se puede dividir un programa en un instante dado*. La ejecución de un programa en un ordenador paralelo puede utilizar un número diferente de procesadores en diferentes periodos de tiempo. Para cada periodo de tiempo, el número de procesadores que se puede llegar a usar para ejecutar el programa se define como el *grado de paralelismo (GDP)*.

A la figura 25, que muestra el GDP en función del tiempo, se le denomina perfil de paralelismo de un programa dado. Por simplicidad, nos concentraremos en el análisis de los perfiles de un único programa. En la figura se muestra un ejemplo de perfil del paralelismo del algoritmo divide y vencerás.

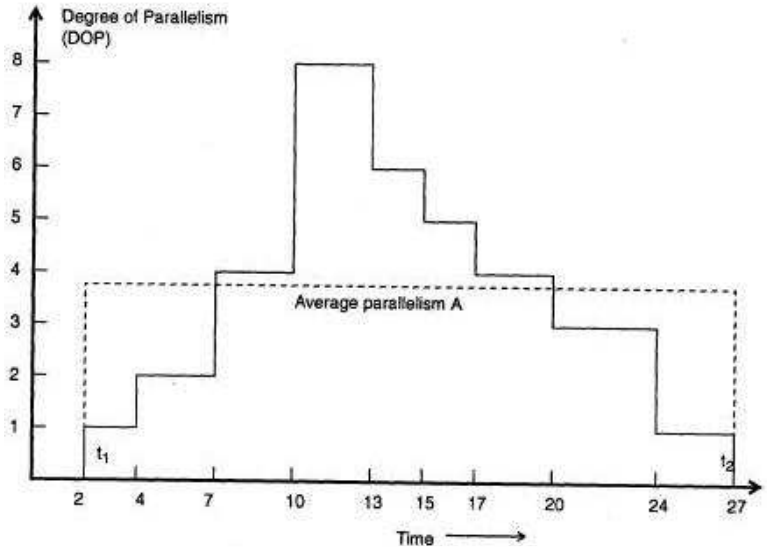

Figura 25 - Perfil del paralelismo de un algoritmo del tipo divide y vencerás

Las fluctuaciones en el perfil durante un periodo de observación depende de la estructura del algoritmo, la optimización del programa, la utilización de recursos, y las condiciones de ejecución del sistema donde se ejecuta el programa.

Paralelismo medio. Consideremos un procesador paralelo compuesto por *n* elementos de proceso homogéneos. Llamamos *m* al paralelismo máximo en un perfil. En el caso ideal $n >> m$. Llamamos Δ a la capacidad de cómputo de un procesador, expresada en MIPS o Mflops, sin considerar las penalizaciones debidas al acceso a memoria, latencia de las comunicaciones, o sobrecarga del sistema. Cuando i procesadores están ocupados durante un periodo de tiempo, se tiene que *GDP = i* en ese periodo.

La capacidad de trabajo realizado, a la que llamaremos *W*, es proporcional al área bajo la curva de perfil paralelo:

$$W = \Delta \int_{t_1}^{t_2} GDP(t)dt$$

Esta integral se calcula frecuentemente mediante el siguiente sumatorio:

$$W = \Delta \sum_{i=1}^{m} i \cdot t_i$$

donde t_i es el tiempo que *GDP=i* y $\sum_{i=1}^{m} t_i = t_2 - t_1$ es el tiempo total de ejecución.

El *paralelismo medio*, que llamaremos *A*, será por tanto

$$A = \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} GDP(t)dt$$

o en su forma discreta

$$A = \frac{\sum_{i=1}^{m} i \cdot t_i}{\sum_{i=1}^{m} t_i}$$

***Speed-up* asintótico**. Si denotamos por $W_i = i\Delta t_i$ al trabajo realizado cuando GPD=i, entonces podemos escribir $W = \sum_{i=1}^{m} W_i$. Esto está suponiendo que no hay sobrecarga de ningún tipo, es decir, se trata del caso ideal de paralelización.

El tiempo de ejecución de W_i sobre un único procesador es $t_i(1) = W_i / \Delta$. El tiempo de ejecución de W_i sobre k procesadores es $t_i(k) = W_i / k\Delta$. Con un número infinito de procesadores disponibles, $t_i(\infty) = W_i / i\Delta$, para *1≤i≤m*. Así, podemos escribir el *tiempo de respuesta* para un procesador de infinitos procesadores como:

$$T(1) = \sum_{i=1}^{m} t_i(1) = \sum_{i=1}^{m} \frac{W_i}{\Delta}$$

$$T(\infty) = \sum_{i=1}^{m} t_i(\infty) = \sum_{i=1}^{m} \frac{W_i}{i\Delta}$$

El *speed-up asintótico* S_∞ se define como el cociente de *T(1)* y $T(\infty)$, es decir, es un parámetro que mide la aceleración del tiempo de cálculo por el hecho de poder paralelizar al máximo la aplicación:

$$S_\infty = \frac{T(1)}{T(\infty)} = \frac{\sum_{i=1}^{m} W_i}{\sum_{i=1}^{m} \frac{W_i}{i}}$$

Si comparamos esta fórmula con la de paralelismo medio, A, se observa que $S_\infty = A$ en el caso ideal. En general, $S_\infty \leq A$ si se consideran las latencias debidas a las comunicaciones y otras sobrecargas del sistema. Observar que tanto S_∞ como A están definidas bajo la suposición de que n=∞ o n>>m.

Paralelismo disponible. Como ya vimos en las primeras secciones de este tema, existe un amplio grado de paralelismo potencial en los programas. Los

códigos científicos presentan un alto grado de paralelismo, debido al paralelismo inherente de los propios datos. *Manoj Kumar* (1988) indica que en códigos de cálculo intensivo es posible ejecutar de 500 a 3.500 operaciones aritméticas simultáneas. *Nicolau* y *Fisher* (1984) mostraron que con un programa Fortran estándar es posible la ejecución simultánea de 90 instrucciones para arquitecturas *VLIW*. Estos números muestran el lado optimista del paralelismo disponible.

David Wall (1991) indica el límite del *ILP* (paralelismo a nivel de instrucción) es de alrededor de 5, raramente sobrepasando el 7. *Bulter* et al. (1991) indican que cuando se eliminan todas las restricciones el GDP puede exceder 17 instrucciones por ciclo. Si el hardware está perfectamente balanceado es posible conseguir de 2,0 a 5,8 instrucciones por ciclo en un procesador superescalar. Estos números muestran el lado pesimista del paralelismo disponible.

2.12.1 Rendimiento medio armónico. Ley de *Amdahl*

Consideremos un sistema paralelo con n procesadores ejecutando m programas en varios modos con diferentes niveles de rendimiento. Queremos definir el rendimiento medio de este tipo de multiprocesadores. Con una distinción de peso podemos definir una expresión del rendimiento.

Cada modo de ejecución puede corresponder a un tipo de ejecución como por ejemplo procesamiento escalar, vectorial, secuencial o paralela. Cada programa puede ejecutarse mediante una combinación de estos modos. El rendimiento medio armónico proporciona un rendimiento medio sobre la ejecución de un gran número de programas ejecutándose en varios modos.

Antes de obtener dicha expresión, estudiaremos las expresiones de la media aritmética y geométrica obtenidas por James Smith[38] (1988). La velocidad de ejecución R_i para el programa i-ésimo se mide en *MIPS* o *Mflops*.

Media aritmética del rendimiento*[37]*. Sea $\{R_i\}$ el conjunto de los rendimiento de los programas i = 1,2,…,m. La media aritmética del rendimiento se define como

$$R_a = \sum_{i=1}^{m} \frac{R_i}{m}$$

La expresión R_a supone que los m programas tienen el mismo peso (1/m). Si existe una distribución de pesos de los distintos programas $\pi = \{f_i$ para i=1,2,…,m}, definimos la media aritmética ponderada del rendimiento como:

$$R_a^* = \sum_{i=1}^{m} f_i R_i$$

Esta media aritmética es proporcional a la suma de los inversos de los tiempos de ejecución; no es inversamente proporcional a la suma de los tiempos de ejecución. Por lo tanto, la media aritmética falla al representar el tiempo real consumido por los *benchmarks*.

Media geométrica del rendimiento[34]. La media geométrica de la velocidad de ejecución o rendimiento para m programas se define como

$$R_g = \prod_{i=1}^{m} R_i^{1/m}$$

Con una distribución de pesos $\pi = \{f_i$ para i=1,2,…,m}, podemos definir una media geométrica ponderada del rendimiento como:

$$R_g^* = \prod_{i=1}^{m} R_i^{f_i}$$

La media geométrica tampoco capta el rendimiento real, ya que no presenta una relación inversa con el tiempo real. La media geométrica ha sido defendida para el uso con cifras de rendimiento que han sido normalizadas con respecto a una máquina de referencia con la que se está comparando.

Rendimiento medio armónico. Debido a los problemas que presentan la media aritmética y geométrica, necesitamos otra expresión del rendimiento medio basado en la media arítmética del tiempo de ejecución. De hecho, $T_i=1/R_i$, es el tiempo medio de ejecución por instrucción para el programa *i*. La media aritmética del tiempo de ejecución por instrucción se define como

$$T_a = \frac{1}{m}\sum_{i=1}^{m} T_i = \frac{1}{m}\sum_{i=1}^{m} \frac{1}{R_i}$$

La media armónica de la velocidad de ejecución sobre m programas de prueba se define por el hecho de que $R_h = \frac{1}{T_a}$:

$$R_h = \frac{m}{\sum_{i=1}^{m} \frac{1}{R_i}}$$

Con esto, el rendimiento medio armónico está de verdad relacionado con el tiempo medio de ejecución. Si consideramos una distribución de pesos, podemos definir el *rendimiento medio armónico ponderado* como:

$$R_h^* = \frac{m}{\sum_{i=1}^{m} \frac{f_i}{R_i}}$$

***Speed-up* armónico medio**. Otra forma de aplicar el concepto de media armónica es ligar los distintos modos de un programa con el número de procesadores usados. Supongamos un programa (o una carga formada por la combinación de varios programas) se ejecutan en un sistema con *n* procesadores. Durante el periodo de ejecución, el programa puede usar *i = 1,2,…,n* procesadores en diferentes periodos de tiempo.

Decimos que el programa se ejecuta en modo *i* si usamos *i* procesadores. R_i se usa para reflejar la velocidad conjunta de *i* procesadores. Supongamos que $T_1 = 1/R_i = 1/i$ es el tiempo de ejecución usando *i* procesadores con una velocidad de ejecución combinada de $R_i=i$ en el caso ideal.

Supongamos que un programa dado se ejecuta en *n* modos de ejecución con una distribución de pesos $w = \{f_i$ para i=1,2,…,m}. El *speed-up armónico medio ponderado* se define como:

$$S = T_1 / T^* = \frac{1}{(\sum_{i=1}^{m} \frac{f_i}{R_i})}$$

donde $T^* = 1/R_h^*$ es la *media armónica ponderada del tiempo de ejecución* para los *n* nodos de ejecución

La figura 26 muestra el comportamiento del *speed-up* para tres funciones de peso distintas.

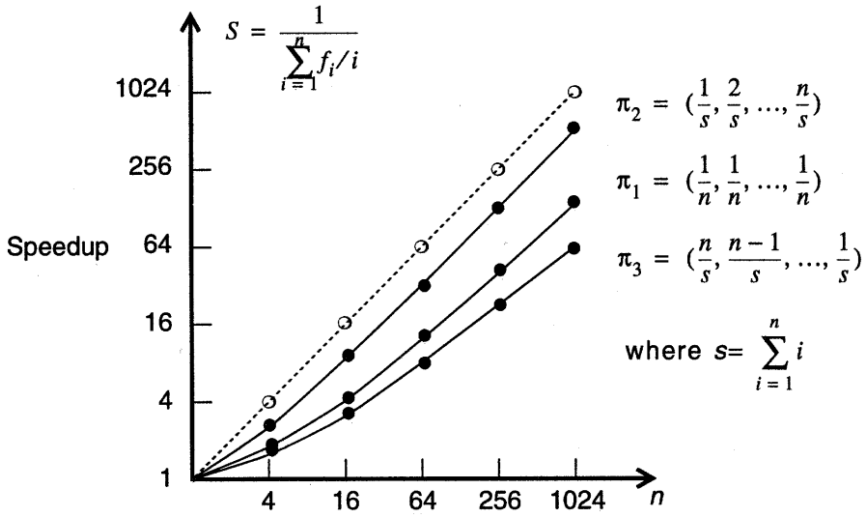

Figura 26 - Media armónica del speed-up con respecto a tres distribuciones de probabilidad: π_1 para la distribución uniforme, π_2 en favor de usar más procesadores y π_3 en favor de usar menos procesadores

Ley de *Amdahl*. De la expresión del *speed-up* armónico medio ponderado de *S* se puede derivar la ley de *Amdahl* como sigue: En primer lugar supongamos que $R_i=i$ y $w=(\alpha,0,0,...,0,1-\alpha)$. Esto implica que el sistema usa un modo secuencial puro con una probabilidad de α, o los *n* procesadores con una probabilidad de *1- α*. Sustituyendo $R_1=1$, $R_n=n$ y w en la ecuación de S, obtenemos la siguiente expresión para el *speed-up*:

$$S_n = \frac{n}{1+(n-1)\alpha}$$

A esta expresión se le conoce como la *ley de Amdahl*. La implicación es que $S \to 1/\alpha$ cuando $n \to \infty$. En otras palabras, independientemente del número de procesadores que se emplee, existe un límite superior del *speed-up* debido a la parte serie de todo programa.

En la figura se han trazado las curvas correspondientes a la ecuación de la ley de *Amdahl* para 4 valores de *α*. El *speed-up* ideal se obtiene para *α=0,* es decir, el caso en que no hay parte serie a ejecutar y todo el código es paralelizable. A poco que el valor de *α* sea no nulo, el *speed-up* máximo empieza a decaer muy deprisa.

Esta ley de *Amdahl* se puede generalizar y es aplicable a cualquier problema que tenga una parte *mejorable* y otra que no se pueda mejorar. Si llamamos F_m a la fracción del problema que se puede mejorar, la fracción del problema que no se puede mejorar será *(1- F_m)*. Dado el problema se tiene una magnitud que es la que mejora con respecto a la inicial, a la magnitud inicial la podemos llamar M_{ini} y a la magnitud una vez aplicadas a las mejoras M_{mej}. La mejora S_{up} es siempre el cociente entre los dos:

$$S_{up} = \frac{M_{ini}}{M_{mej}}$$

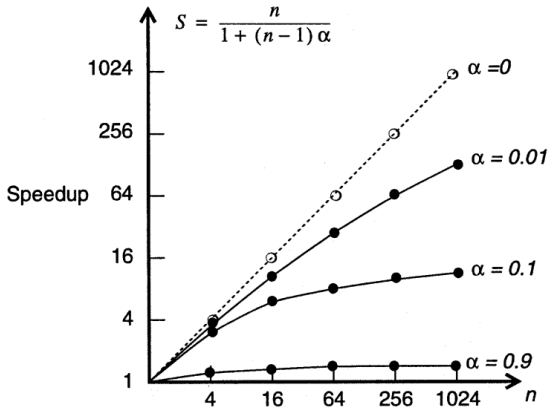

Figura 27 - Mejora del rendimiento para diferentes valores de α, donde α es la fracción del cuello de botella secuencial

Este cociente se puede poner en función de las fracciones que son mejorables y las que no, ya que $M_{ini} = K \cdot ((1 - F_m) + F_m) = K$, y $M_{mej} = K \cdot ((1 - F_m) + F_m / S_m)$, donde la *K* es una proporcionalidad con las unidades de la magnitud del problema, y S_m es el factor de mejora de la parte mejorable. Con todo esto se puede reescribir la mejora total del sistema al intentar mejorar S_m veces la parte mejorable como:

$$S_{up} = \frac{1}{1 - F_m + \frac{F_m}{S_m}}$$

Que es exactamente la misma expresión obtenida aplicando el *speed-up* medio armónico.

La expresión de la ley de *Amdahl* generalizada se puede aplicar a cualquier problema. Por ejemplo, el rendimiento relativo vectorial/escalar en los procesadores vectoriales, no es más que la aplicación de esta ley el caso en que un programa tenga una parte vectorizable (parte que va a mejorar) y otra escalar, cuyo rendimiento no va a mejorar por el hecho de estar utilizando un procesador vectorial.

2.12.2 Modelos del rendimiento del speed-up

En esta sección se describen tres modelos de medición del *speed-up*. La ley de *Amdahl* (1967) se basa en una carga de trabajo fija o en un problema de tamaño fijo. La ley de *Gutafson* (1987) se aplica a problemas escalables, donde el tamaño del problema se incrementa al aumentar el tamaño de la máquina o se dispone de un tiempo fijo para realizar una determinada tarea. El modelo de *speed-up* de *Sun y Ni* (1993) se aplica a problemas escalables limitados por la capacidad de memoria. En la figura 28 se muestra un esquema de los tres modelos utilizados.

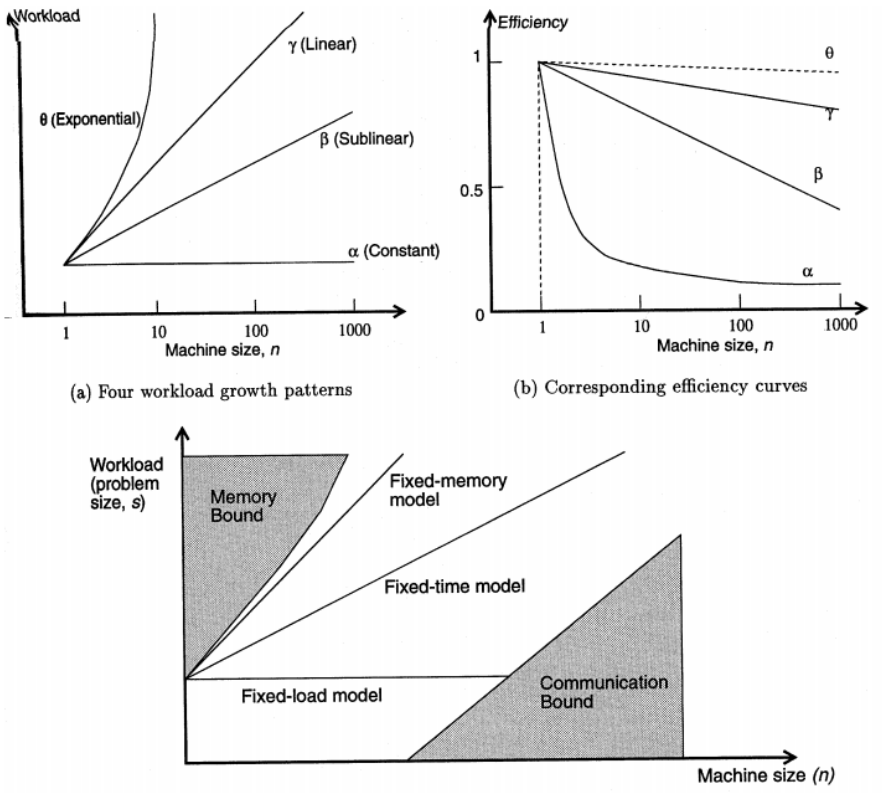

Figura 28 - Modelos de rendimiento del *speed-up*

Ley de *Amdahl*, limitación por carga de trabajo fija

En muchas aplicaciones prácticas, donde es importante la respuesta más rápida posible, la carga de trabajo se mantiene fija y es el tiempo de ejecución lo que se debe intentar reducir. Al incrementarse el número de procesadores en el sistema paralelo, la carga fija se distribuye entre más procesadores para la ejecución paralela. Por lo tanto, el objetivo principal es obtener los resultados lo más pronto posible. En otras palabras, disminuir el tiempo de respuesta es nuestra principal meta. A la ganancia de tiempo obtenida para este tipo de aplicaciones donde el tiempo de ejecución es crítico se le denomina *speed-up bajo carga fija*.

Speed-up bajo carga fija. La fórmula vista en el apartado anterior se basa en una carga de trabajo fija, sin importar el tamaño de la máquina. Las formulaciones tradicionales del *speed-up*, incluyendo la ley de *Amdahl*, están basadas en un problema de tamaño fijo y por lo tanto en una carga fija. En este caso, el factor de *speed-up* está acotado superiormente por el cuello de botella secuencial.

A continuación se considera las dos posibles situaciones: $GDP<n$ ó $GDP\geq n$. Consideremos el caso donde el $GDP=i\geq n$. Supongamos que todos los n procesadores se usan para ejecutar W_i exclusivamente. El tiempo de ejecución de W_i es

$$t_i(n) = \frac{W_i}{i\Delta}\left\lceil \frac{i}{n} \right\rceil$$

De esta manera el tiempo de respuesta es

$$T(n) = \sum_{i=1}^{m} \frac{W_i}{i\Delta}\left\lceil \frac{i}{n} \right\rceil$$

Observar que si *i<n*, entonces $t_i(n) = t_i(\infty) = W_i / i\Delta$. Ahora, definimos el *speed-up para carga fija* como:

$$S_n = \frac{T(1)}{T(n)} = \frac{\sum_{i=1}^{m} W_i}{\sum_{i=1}^{m} \frac{W_i}{i}\left\lceil \frac{i}{n} \right\rceil}$$

Observar que $S_n \le S_\infty \le A$.

Existe una gran cantidad de factores que pueden rebajar el *speed-up*. Estos factores incluyen latencias de comunicaciones debidas a retrasos en el acceso a la memoria, comunicaciones a través de un bus o red, o sobrecarga del sistema operativo y retrasos causados por las interrupciones. Si *Q(n)* es la suma de todas las sobrecargas del sistema en un sistema con *n* procesadores, entonces:

$$S_n = \frac{T(1)}{T(n) + Q(n)} = \frac{\sum_{i=1}^{m} W_i}{\sum_{i=1}^{m} \frac{W_i}{i}\left\lceil \frac{i}{n} \right\rceil + Q(n)}$$

El retraso por sobrecarga *Q(n)* depende siempre de la aplicación y de la máquina. Es muy difícil obtener una expresión para *Q(n)*. A no ser de que especifique otra cosa se supondrá que *Q(n)=0* para simplificar la explicación.

Ley de Amdahl revisada. En 1967, *Gene Amdahl* derivó un speed-up para el caso particular donde el computador opera en modo puramente secuencial (*GDP=1*) o en modo totalmente paralelo (*GDP=n*). Es decir, $W_i=0$ si $i \neq n$. En este caso, el *speed-up* viene dado por:

$$S_n = \frac{W_1 + W_n}{W_1 + W_n / n}$$

La ley de *Amdahl* supone que la parte secuencial del programa W_i no cambia con respecto a tamaño *n* de la máquina. Sin embargo, la porción paralela se ejecuta equitativamente por los *n* procesadores reduciéndose el tiempo de ejecución de esta parte.

Suponiendo una situación normalizada en la cual $W_1=\alpha$ y $W_n=1-\alpha$ se tiene que $W_1+W_n= \alpha+1- \alpha=1$. Con esta situación, las anteriores ecuaciones son la misma. Igual que en aquella expresión α es la fracción serie del programa y (1- α) la paralelizable.

La ley de *Amdahl* se ilustra en la figura 29. Cuando el número procesadores aumenta, la carga ejecutada en cada procesador decrece. Sin embargo, la cantidad total de trabajo (carga) W_1+W_n se mantiene constante como se muestra en la figura 29a. En la figura 29b, el tiempo total de ejecución decrece porque $T_n = W_n / n$. Finalmente, el término secuencial domina el rendimiento porque $T_n \to 0$ al hacer *n* muy grande siendo T_1 constante.

Cuello de botella secuencial. La figura 29c muestra una gráfica de la ley de Amdahl para diferentes valores de $0 \le \alpha \le 1$. El máximo *speed-up*, $S_n=n$, se obtiene para $\alpha=0$. El mínimo *speed-up*, $S_n=1$, se obtiene para $\alpha=1$. Cuando $n \to \infty$, el valor límite es $S_\infty = 1/\alpha$. Esto implica que el *speed-up* está acotado superiormente para 1/ α, independientemente del tamaño de la máquina.

La curva del *speed-up* en la figura 29c cae rápidamente al aumentar α. Esto significa que con un pequeño porcentaje de código secuencial, el rendimiento total no puede ser superior a 1/ α. A este α se le denomina *cuello de botella secuencial* de un programa.

El problema de un cuello de botella secuencial no puede resolverse incrementando el número de procesadores del sistema. El problema real está en la existencia de una fracción secuencial *(s)* del código. Esta propiedad ha impuesto una visión muy pesimista del procesamiento paralelo en las pasadas dos décadas.

De hecho, se observaron dos impactos de esta ley en la industria de los computadores paralelos. En primer lugar, los fabricantes dejaron de lado la construcción de computadores paralelos de gran escala. En segundo lugar, una parte del esfuerzo investigador se desplazó al campo de desarrollo de compiladores paralelos en un intento de reducir el valor de α y mejorar de esa forma el rendimiento.

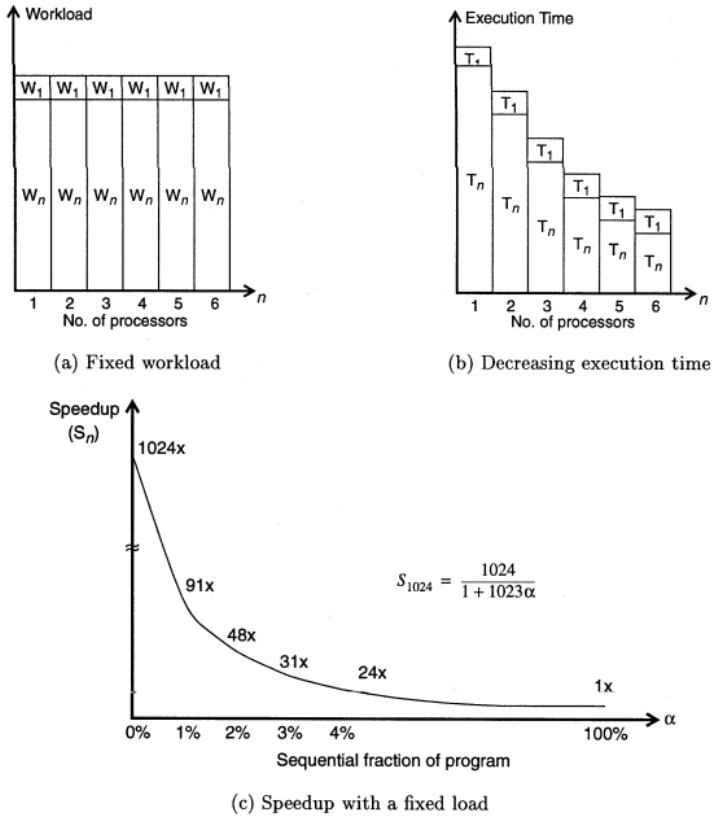

Figura 29 - Modelo del *speed-up* de carga fija y la ley de *Amdahl*

La ley de *Gustafson*, limitación por tiempo fijo.

Uno de los mayores inconvenientes de aplicar la ley de *Amdahl* es que el problema (la carga de trabajo) no puede aumentarse para corresponderse con el poder de cómputo al aumentar el tamaño de la máquina. En otras palabras, el tamaño fijo impide el escalado del rendimiento. Aunque el cuello de botella secuencial es un problema importante, puede aliviarse en gran medida eliminando la restricción de la carga fija (o tamaño fijo del problema). John *Gustafson* (1988)[35] ha propuesto el concepto de tiempo fijo que da lugar a un modelo del *speed-up* escalado.

Escalado para conseguir una mayor precisión. Las aplicaciones de tiempo real son la causa principal del desarrollo de un modelo de *speed-up* de carga fija y la ley de *Amdahl*. Existen muchas otras aplicaciones que enfatizan la precisión más que el tiempo de respuesta. Al aumentar el tamaño de la máquina para obtener una mayor potencia de cálculo, queremos incrementar el tamaño del problema para obtener una mayor carga de trabajo produciendo una solución más precisa y manteniendo el tiempo de ejecución.

Un ejemplo de este tipo de problemas es el cálculo de la predicción meteorológica. Habitualmente se tiene un tiempo fijo para calcular el tiempo que hará en unas horas, naturalmente se debe realizar el cálculo antes de que la lluvia llegue. Normalmente se suele imponer un tiempo fijo de unos 45

minutos a una hora. En ese tiempo se tiene que obtener la mayor precisión posible. Para calcular la predicción se sigue un modelo físico que divide el área terrestre a analizar en cuadrados de manera que los cálculos realizados en cada uno de estos cuadrados se puede hacer en paralelo. Si se disponen de muchos procesadores se podrán hacer cuadros más pequeños con lo que la precisión de aumenta manteniendo el tiempo de ejecución.

***Speed-up* de tiempo fijo**. En aplicaciones de precisión crítica, se desea resolver un problema de mayor tamaño en una máquina mayor con aproximadamente el mismo tiempo de ejecución que costaría resolver un problema menor en una máquina menor. Al aumentar el tamaño de la máquina, tendremos una nueva carga de trabajo y por lo tanto un nuevo perfil del paralelilsmo. Sea *m'* el máximo GDP con respecto al problema escalado y W_i' la carga de trabajo con *GDP=i*.

Observar que, en general, $W_i' > W_i$ para $2 \le i \le m'$ y $W_1' = W_1$. El *speed-up* de tiempo fijo se define bajo el supuesto de que $T(1) = T'(n)$, donde $T_n'(n)$ es el tiempo de ejecución del problema escalado y *T(1)* se corresponde con el problema original sin escalar. Así, tenemos que

$$\sum_{i=1}^{m} W_i = \sum_{i=1}^{m} \frac{W_i'}{i} \left\lceil \frac{i}{n} \right\rceil + Q(n)$$

Una fórmula general para el *speed-up* de tiempo fijo se define por $S_n' = T'(1)/T'(n) = T'(1)/T(1)$. Por analogía con la ecuación $S_n = \dfrac{T(1)}{T(n)+Q(n)} = \dfrac{\sum_{i=1}^{m} W_i}{\sum_{i=1}^{m} \frac{W_i}{i}\left\lceil \frac{i}{n} \right\rceil + Q(n)}$ se obtiene la expresión para el *speed-up* de tiempo fijo:

$$S_n' = \frac{\sum_{i=1}^{m'} W_i'}{\sum_{i=1}^{m} \frac{W_i'}{i}\left\lceil \frac{i}{n} \right\rceil + Q(n)} = \frac{\sum_{i=1}^{m'} W_i'}{\sum_{i=1}^{m} W_i}$$

Ley de Gustafson[35]. El *speed-up* de tiempo fijo fue desarrollado por *Gustafson* para un perfil de paralelismo especial con $W_i=0$ si $i \neq n$. De forma similar a la ley de Amdahl, podemos reescribir la ecuación anterior como sigue (suponemos *Q(n)=0*):

$$S_n' = \frac{\sum_{i=1}^{m'} W_i'}{\sum_{i=1}^{m} W_i} = \frac{W_1' + W_n'}{W_1 + W_n} = \frac{W_1 + nW_n}{W_1 + W_n}$$

La figura 30a muestra la relación del escalado de la carga de trabajo con el speed-up escalado de Gustafson. De hecho, la ley de Gustafson puede reformularse en términos de $\alpha = W_1$ y $1 - \alpha = W_n$, bajo la suposición de que $W_1+W_n=1$, como sigue:

$$S_n' = \frac{\alpha + n(1-\alpha)}{\alpha + (1-\alpha)} = n - \alpha(n-1)$$

Obsérvese que la pendiente de la curva S_n en la figura 30c es mucho más plana que en la figura 29c. Esto implica que la ley de Gustafson soporta el rendimiento escalable al aumentar el tamaño de la máquina. La idea es mantener a todos los procesadores ocupados incrementando el tamaño del problema.

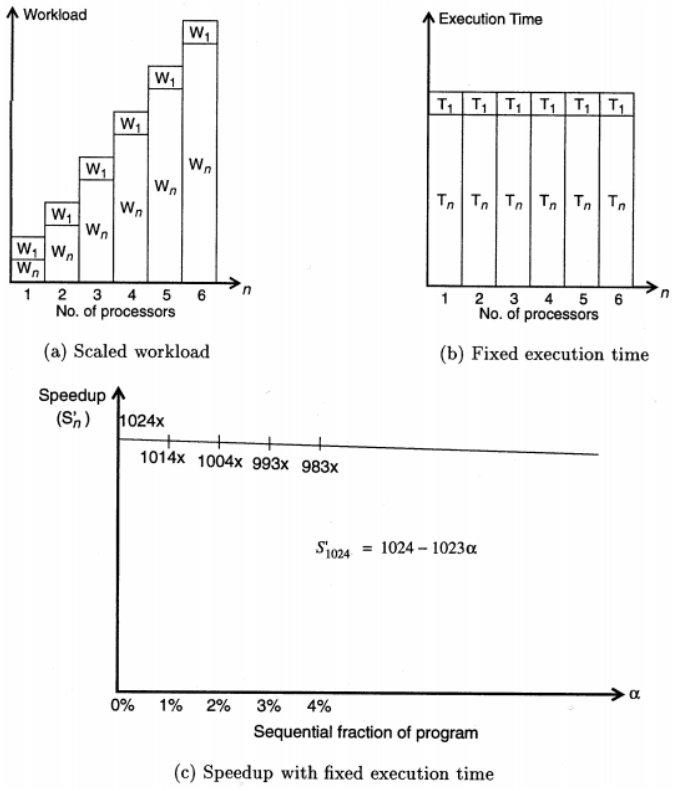

Figura 30 - Modelo de *speed-up* de tiempo fijo y la ley de Gustafson

Modelo del *speed-up* limitado por la memoria fija

Xian-He Sun y Lionel Ni (1993) han desarrollado un modelo del *speed-up* limitado [36] por la memoria que generaliza la ley de Amdahl y Gustafson para maximizar el uso de la CPU y la memoria. La idea es resolver el mayor problema posible, limitado por el espacio de memoria. En este caso también es necesario una carga de trabajo escalada, proporcionando un mayor *speed-up*, mayor precisión y mejor utilización de los recursos.

Problemas limitados por el espacio de memoria. Los cálculos científicos y las aplicaciones de ingeniería suelen necesitar una gran cantidad de memoria. De hecho, muchas aplicaciones de los ordenadores paralelos surgen de la limitación de la memoria más que de la CPU o la E/S. Esto es especialmente cierto en sistemas multicomputador con memoria distribuida. Cada elemento de proceso está limitado a usar su propia memoria local por lo que sólo puede hacer frente a un pequeño subproblema.

Cuando se utiliza un mayor número de nodos para resolver un problema grande, la capacidad de memoria total se incrementa de forma proporcional. Esto le permite al sistema resolver un problema escalado mediante el particionamiento del programa y la descomposición del conjunto de datos.

En lugar de mantener fijo el tiempo de ejecución, uno puede querer usar toda la memoria disponible para aumentar aún más el tamaño del problema. En otras palabras, si se tiene un espacio de memoria adecuado y el problema escalado cumple el límite de tiempo impuesto por la ley de Gustafson, se puede incrementar el tamaño del problema, consiguiendo una mejor solución o una solución más precisa.

El modelo de limitación de la memoria se desarrolló bajo esta filosofía. La idea es resolver el mayor problema posible limitado únicamente por la capacidad de memoria disponible.

Speed-up de memoria fija. Sea *M* el requisito de memoria para un problema dado y *W* la carga computacional. Ambos factores están relacionados de varias formas, dependiendo del direccionamiento del espacio y las restricciones de la arquitectura. Así, podemos escribir $W = g(M)$ o $M = g^{-1}(W)$.

En un multicomputador, y en la mayoría de multiprocesadores, la capacidad total de la memoria se incrementa linealmente con el número de nodos disponibles. Sea $W = \sum_{i=1}^{m} W_i$ la carga para una ejecución secuencial del programa en un único nodo, y $W^* = \sum_{i=1}^{m} W_i^*$ la carga para el problema de los *n* nodos, donde m^* es el máximo GDP del problema escalado. Los requisitos de memoria de un nodo activo está limitado por $M = g^{-.1}(\sum_{i=1}^{m} W_i)$.

El speed-up con memoria fija se define de forma similar al caso de la ecuación

$$S_n' = \frac{\sum_{i=1}^{m'} W_i'}{\sum_{i=1}^{m} \frac{W_i'}{i} \left\lceil \frac{i}{n} \right\rceil + Q(n)} = \frac{\sum_{i=1}^{m'} W_i'}{\sum_{i=1}^{m} W_i} :$$

$$S_n^* = \frac{\sum_{i=1}^{m} W_i^*}{\sum_{i=1}^{m^*} \frac{W_i^*}{i} \left\lceil \frac{i}{n} \right\rceil + Q(n)}$$

La carga de trabajo para la ejecución secuencial en un único procesador es independiente del tamaño del problema o del tamaño del sistema. Así, podemos escribir $W_1 = W_1' = W_1^*$ para los tres modelos de *speed-up*. Consideremos el caso especial con dos modos de operación: ejecución secuencial frente a perfectamente paralela. La mejora donde *nM* es el incremento en la capacidad de la memoria para un multicomputador con *n* nodos.

Supongamos además que $g^*(nM) = G(n)g(M) = G(n)W_n$ donde $W_n = (M)$ y g^* es una función homogénea. El factor *G(n)* refleja el incremento en la carga al aumentar la memoria *n* veces. Esto nos permite rescribir la fórmula anterior bajo la suposición de que $W_i = 0$ si *i≠1* 0 *n* y *Q(n)=0*:ç

$$S_n^* = \frac{W_1^* + W_n^*}{W_1^* + W_n^* / n} = \frac{W_1 + G(n)W_n}{W_1 + G(n)W_n / n}$$

Rigurosamente hablando este modelo sólo es válido bajo estas dos suposiciones: (1) El conjunto de toda la memoria forma un espacio global de direcciones (en otras palabras, suponemos un espacio de memoria compartido distribuido); (2) Todo el espacio de memoria disponible se utiliza para el problema escalado. Existen tres casos especiales donde se puede aplicar la ecuación:

- *G(n)=1*. Se corresponde con el caso donde el tamaño del problema es fijo, siendo equivalente a la ley de Amdahl.

- *G(n) = n*. Se aplica al caso en el que la carga se incrementa n veces cuando la memoria se incrementa n veces. En este caso, la ecuación se corresponde con la ley de Gustafson con un tiempo de ejecución fijo.

- *G(n) > n*. Se corresponde con la situación donde la carga computacional se incrementa más rápidamente que los requisitos de memoria. En este caso, el modelo de la memoria fija da posiblemente todavía un mayor *speed-up* que el de tiempo fijo.

De este análisis se pueden obtener las siguientes conclusiones: la ley de Amdahl y la ley de Gustafson son casos particulares del modelo fijo. Cuando la computación crece más rápidamente que los requisitos de memoria, lo que es frecuente en el caso de algunas simulaciones científicas y aplicaciones de ingeniería, el modelo de memoria fija (figura 31) da lugar a un mayor speed-up (es decir, $S_n^* \geq S_n' \geq S_n$) y una mejor utilización de los recursos.

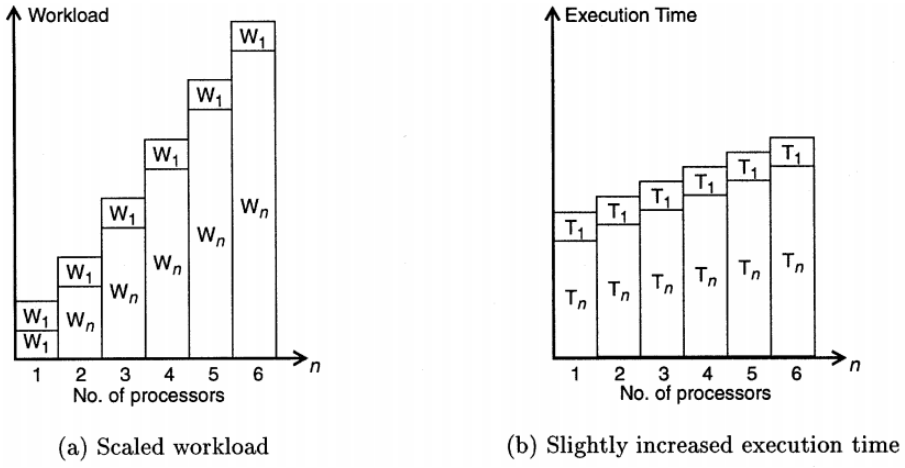

Figura 31 - Modelo de *speed-up* de memoria fija

3. Implementación Paralelización MPI

En el presente capítulo se pretende tratar en profundidad la técnica de implementación seguida de la paralelización del sistema empleando la librería de MPI basada en el paso de mensajes.

Antes de poder realizar la paralelización del sistema, se tuvo que analizar exhaustivamente el código de la versión secuencial para comprender perfectamente el funcionamiento de éste y así poder identificar aquellas zonas de código susceptibles de paralelizarse. A continuación se detalla el razonamiento seguido para definir la estrategia de paralelización.

Descomposición del problema

Esta primera etapa se centra en la definición de pequeñas tareas que conformasen el algoritmo con el fin de convertirlo en grano fino. A la hora de estudiar dicha descomposición, se ha tenido en cuenta que la estrategia de paralelización a seguir debe reducir al máximo la duplicación de datos y computación.

En la descomposición del problema, se han considerado dos enfoques diferentes para poder encontrar la alternativa más óptima. El primero de los enfoques, ha sido centrar la atención en qué cómputos son potencialmente paralelizables, el segundo enfoque empleado, ha sido centrar dicha atención en qué partición de los datos es necesaria realizar en función de los cómputos anteriormente definidos como paralelizables.

La estrategia elegida, ha sido convertir el algoritmo en un algoritmo maestro-esclavo ya que cumplía una serie requisitos que se tuvieron en cuenta en el diseño de la paralelización:

- Se necesitaba que el comportamiento del algoritmo paralelo fuese idéntico al algoritmo secuencial. Más adelante, se comentaran otras posibilidades de paralelización correctas, pero que debido a la lógica del funcionamiento, tenían un comportamiento del algoritmo diferente lo cual implica en una repercusión en los resultados obtenidos por el algoritmo.

- El número de individuos empleados en los algoritmos genéticos generalmente, siempre serán muy superiores al número de procesadores disponibles.

- La generación de cada individuo y las operaciones de definición del *fitness* y resto de operaciones a realizar con el individuo, son totalmente independientes del resto de individuos.

- El repartir los individuos adecuadamente entre los procesadores a emplear se traducían en tareas de tamaños equivalentes, ya que las operaciones a realizar con cada individuo es constante, de esta forma, se facilita el balanceo de la carga de los procesadores.

- El aumento de procesadores, podrá ser aprovechado perfectamente con esta estrategia ya que convierte el algoritmo en escalable.

Posteriormente, se pensó en una vez definida la estrategia de maestro-esclavo, qué datos debían ser transferidos o compartidos entre las distintas tareas.

La estrategia de paralelizar el sistema mediante un algoritmo maestro-esclavo, fue consecuencia de realizar un análisis de las distintas alternativas posibles. A continuación, se muestran otras alternativas de paralelización posibles:

1. **Algoritmo paralelizado mediante grano fino**: en esta estrategia de paralelización la población de cada una de las generaciones se encuentra dividida entre los distintos procesadores, e idealmente, cada procesador debería albergar un único individuo. Cada procesador procesa su subpoblación. Los algoritmos de grano fino dividen la población en pequeñas subpoblaciones que contienen sólo una o dos soluciones que están conectadas en topología de rejilla. El cruce y la selección de individuos se hará entre los individuos que pertenezcan a un mismo vecindario, formado por un conjunto de individuos adyacentes según la representación espacial antes citada. Se podría permitir el solapamiento entre vecindario para propiciar la interacción, aunque leve, entre todos los individuos de la población.

   El propósito original de los algoritmos genéticos de grano fino era usar un gran número de pequeños procesadores, donde cada procesador procesaría una única solución. En dicha implementación, la población de soluciones candidato se ha acotado a una rejilla bidimensional, donde cada posición de la rejilla puede contener una solución particular a estar vacía. En cada iteración del algoritmo, todas las soluciones pasan por el operadro de mutación, en el que el individuo se combina con un vecino escogido al azar de su vecindario. El individuo, pasa entonces por un operador de mutación que realiza una ligera modificación a la solución actual. Si el nuevo individuo es de una calidad mayor, entonces reemplaza al individuo original.

   La versión secuencial del sistema analiza toda la población de individuos en cada una de las generaciones y aplica el algoritmo evolutivo elegido. En la implementación de la paralelización mediante grano fino, se ha podido comprobar que el comportamiento no sería el mismo puesto que la población se divide entre distintos procesadores y se aplica posteriormente el algoritmo evolutivo entre los procesadores vecinos, por lo que, no se trabaja con el total de la población. Además, se trata de una implementación

2. **Algoritmo paralelizado mediante grano grueso**: en este caso se realiza uso de múltiples poblaciones y migración de individuos entre

ellas. Dado que cada una de las poblaciones evolucionan independientemente, el ratio de migración es muy importante de cara a obtener resultados satisfactorios. Conociendo como migración al intercambio de individuos entre subpoblaciones.

Al igual que en el caso del algoritmo de grano fino, el comportamiento de la versión secuencial del sistema se vería alterado ya que el algoritmo trabaja sobre múltiples poblaciones.

3. **Algoritmo paralelizado mediante un híbrido**: estrategia que combina la aplicación de algoritmos de grano fino con los de grano grueso.

   Por consecuencia, mediante la combinación, el comportamiento continúa siendo diferente a la versión secuencial.

Aplicados los puntos anteriores, a continuación se detalla la descomposición del problema.

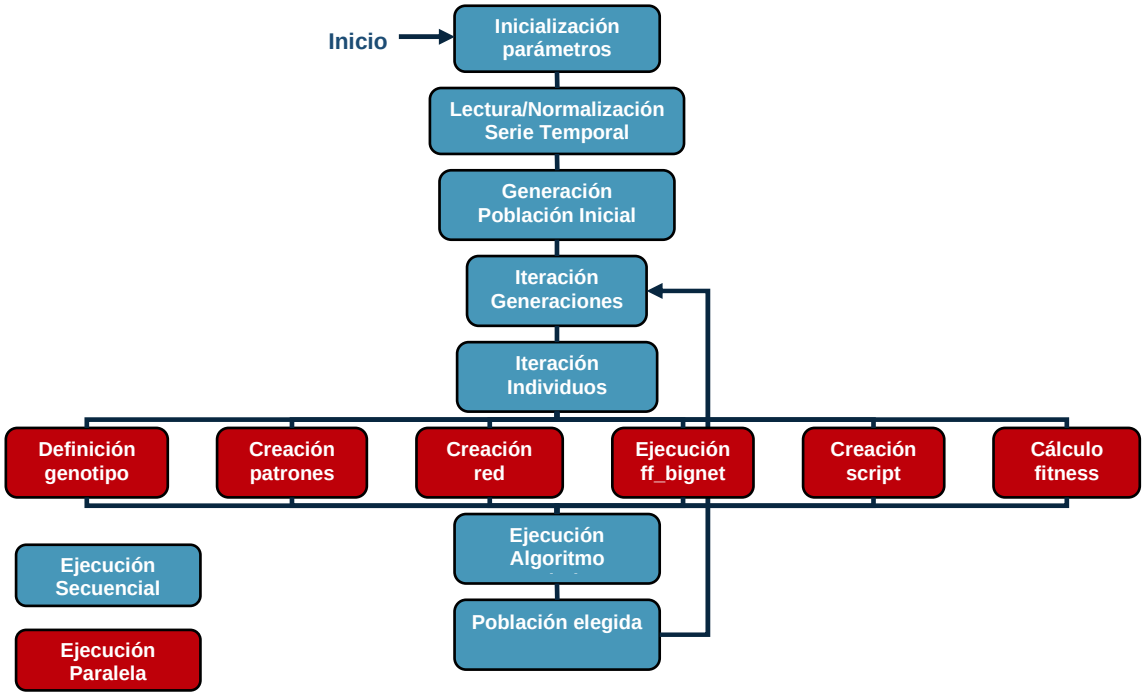

Figura 32 - Diseño de la descomposición de la paralelización del algoritmo

Inicialmente, el algoritmo realiza la inicialización de las variables del experimento leyendo el fichero de configuración que se pasa como parámetro en la ejecución. Además, se realiza la lectura de la serie temporal pasada como parámetro en la ejecución y su posterior normalización. A continuación, se genera la población inicial.

Dichas tareas, tiene sentido que las realice el nodo maestro, puesto que el pasar a paralelizar dichas tareas significarían una gran cantidad de envío de mensajes, ya que como se verá a continuación, tanto la configuración del experimento, como la serie temporal como la población inicial, debe ser conocida por todos los procesadores que intervengan en la ejecución.

Posteriormente, se pasa a la iteración de de los individuos donde ahora sí, se pasa a ejecutar de forma paralela cada una de las tareas que se realiza sobre cada un de los individuos. Tal y como se ha comentado anteriormente, dichas tareas sobre cada individuo son independientes entre los procesadores, por lo que, no es necesario realizar ningún paso de mensajes. Al finalizar la iteración de individuos, el nodo maestro deberá recibir el resultado de todos los individuos para poder pasar a ejecutar el algoritmo genético pasado por parámetro.

Con la ejecución de dicho parámetro, se construirá la población elegida que pasará a ser la base de la nueva iteración de la generación.

Fase de comunicación

Una vez definida la estrategia de implementación del sistema descrita en la fase de descomposición, se debe analizar qué datos van a ser tratados por cada uno de los procesadores y definir la estrategia de comunicación.

En esta fase se ha definido qué datos son necesarios compartir o transferir, para que el algoritmo se pueda ejecutar en paralelo. Se ha observado, que existe una serie de datos que son imprescindibles para poder llevar a cabo el algoritmo, como era la configuración definida de ejecución, pasada en un fichero de entrada, y la población inicial del algoritmo.

A la hora de diseñar la estrategia de comunicación, se ha tenido en cuenta que los mensajes a trasmitir deben ser lo más pequeños posibles, para poder lograr así reducir al máximo la penalización en tiempo la ejecución debido a los distintos pasos de mensajes a realizar. Además, se ha considerado que las operaciones de comunicación a realizar se pudieran realizar concurrentemente.

Para poder realizar los envíos de los datos de la configuración del experimento, ha sido necesario, la creación de tipos de datos compuestos. A continuación, se detalla cada uno de los tipos compuestos creados:

```
void buildTypeDerivedGeneracion (t_gen *gen, MPI_Datatype*  generacion){
  int blockLengths [11] = {1,1,1,1,1,1,1,1,1,1,1};
  MPI_Datatype types[11] = {MPI_INT, MPI_INT, MPI_INT, MPI_INT, MPI_INT, MPI_INT, MPI_INT, MPI_INT, MPI_INT,
MPI_FLOAT, MPI_FLOAT};
  MPI_Aint displacements[11];
  /* Use for calculating displacements              */
  MPI_Aint start_address;
  MPI_Aint address;
  displacements[0] = 0;

  MPI_Address(&(gen->population_size), &start_address);
  MPI_Address(&(gen->mutation_probability), &address);
  displacements[1] = address - start_address;
  MPI_Address(&(gen->max_num_generations), &address);
  displacements[2] = address - start_address;
  MPI_Address(&(gen->selection_mode), &address);
  displacements[3] = address - start_address;
  MPI_Address(&(gen->chrom_lenght), &address);
  displacements[4] = address - start_address;
  MPI_Address(&(gen->type_chrom), &address);
  displacements[5] = address - start_address;
  MPI_Address(&(gen->shuffle), &address);
  displacements[6] = address - start_address;
  MPI_Address(&(gen->crossvalidation_number_subsets), &address);
  displacements[7] = address - start_address;
  MPI_Address(&(gen->evolutionary_algorithm), &address);
  displacements[8] = address - start_address;
  MPI_Address(&(gen->cr), &address);
  displacements[9] = address - start_address;
  MPI_Address(&(gen->factor), &address);
  displacements[10] = address - start_address;

  /* Build the derived datatype */
    MPI_Type_struct(11, blockLengths, displacements,
       types, generacion);

    /* Commit it -- tell system we'll be using it for */
    /* communication.                              */
    MPI_Type_commit(generacion);
}
```

Figura 32 Tipo de datos derivado de la generación

La necesidad de construir tipos de datos derivados surge ya que el tipo de datos empleados en la estructura no es un tipo de datos básicos de MPI, por lo que se necesita construir un tipo de datos MPI a partir del tipo de datos de C. MPI propone una solución a esto permitiendo al usuario construir tipos de datos MPI en tiempo de ejecución. Para construir un tipo de datos MPI básicamente se especifica la distribución de los datos en el tipo (los tipos de los miembros y sus direcciones relativas de memoria).

Para construir los tipos derivados, se deben traducir todos los elementos que forman parte del tipo derivado en tipos básicos MPI, por lo que, en el array *MPI_Datatype types* se almacenan los tipos básicos MPI que van a conformar el tipo básico.

Para poder construirlos, se ha empleado la función MPI_Address() que ayuda a calcular los desplazamientos de cada uno de los miembros con respecto a la dirección inicial del primero.

```
void buildTypeDerivedParametrosExperimento(t_exp_parameters *exp_parameters, MPI_Datatype*  parametrosExperimento){

int blockLengths [18] = {1,MAX_LONG_FILE_NAME,MAX_LONG_FILE_NAME,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1};
MPI_Datatype types[18] = {MPI_INT, MPI_CHAR, MPI_CHAR, MPI_INT, MPI_DOUBLE, MPI_DOUBLE, MPI_INT,
MPI_DOUBLE, MPI_DOUBLE, MPI_INT,MPI_INT,MPI_INT,MPI_INT,MPI_INT,MPI_INT,MPI_INT,MPI_INT,MPI_INT};
MPI_Aint displacements[18];
/* Use for calculating displacements            */
MPI_Aint start_address;
MPI_Aint address;
displacements[0] = 0;
MPI_Address(&(exp_parameters->num_ex), &start_address);
MPI_Address(&(exp_parameters->parameter_file_name), &address);
displacements[1] = address - start_address;
MPI_Address(&(exp_parameters->timeserie_file_name), &address);
displacements[2] = address - start_address;
MPI_Address(&(exp_parameters->precision), &address);
displacements[3] = address - start_address;
MPI_Address(&(exp_parameters->maximum), &address);
displacements[4] = address - start_address;
MPI_Address(&(exp_parameters->minimum), &address);
displacements[5] = address - start_address;
MPI_Address(&(exp_parameters->prct_increment), &address);
displacements[6] = address - start_address;
MPI_Address(&(exp_parameters->max_for_norm), &address);
displacements[7] = address - start_address;
MPI_Address(&(exp_parameters->min_for_norm), &address);
displacements[8] = address - start_address;
MPI_Address(&(exp_parameters->seed), &address);
displacements[9] = address - start_address;
MPI_Address(&(exp_parameters->norm_mode), &address);
displacements[10] = address - start_address;
MPI_Address(&(exp_parameters->subset_generation_mode), &address);
displacements[11] = address - start_address;
MPI_Address(&(exp_parameters->prct_train), &address);
displacements[12] = address - start_address;
MPI_Address(&(exp_parameters->prct_test), &address);
displacements[13] = address - start_address;
MPI_Address(&(exp_parameters->prct_validation), &address);
displacements[14] = address - start_address;
MPI_Address(&(exp_parameters->prct_fitness_test), &address);
displacements[15] = address - start_address;
MPI_Address(&(exp_parameters->prct_fitness_validation), &address);
displacements[16] = address - start_address;
MPI_Address(&(exp_parameters->cv_set), &address);
displacements[17] = address - start_address;

MPI_Type_struct(18, blockLengths, displacements,
types, parametrosExperimento);

MPI_Type_commit(parametrosExperimento);
}
```

Figura 33 Tipo derivado parámetros experimento

Tal y como se comentó anteriormente, el nodo maestro es el encargado de realizar la lectura del fichero de parámetros del experimento (*epf_nn3.101.txt*). Posteriormente, cada uno de los procesadores que intervienen en la paralelización, deben disponer de dicha información, por lo que, antes de iniciar la paralelización, el nodo maestro debe realizar el envío de todos estos parámetros. El tipo de datos derivado de los parámetros del experimento, permite realizar este envío por parte del nodo maestro.

```
void buildTypeDerivedPopulation (gene population_aux[MAX_NUMBER_INDIVIDUALS][MAX_NUM_GENES], MPI_Datatype*
poblacion){
  MPI_Type_vector(MAX_NUMBER_INDIVIDUALS, MAX_NUM_GENES, MAX_NUM_GENES, MPI_CHAR, poblacion);
  /* Commit it -- tell system we'll be using it for */
  /* communication.                              */
  MPI_Type_commit(poblacion);
}
```

Figura 34 Tipo derivado de la población

Posteriormente, puesto que la primera generación del experimento es generada de forma aleatoria por parte del nodo maestro, o en el caso, que se desee realizar la lectura de la última población del fichero escrito en cada generación (*last_generation.txt*) también es necesario realizar un envío por parte del nodo maestro al resto de procesadores para que dispongan la información de los individuos de la generación de partida. Para ello, se construye el tipo derivado de la población.

En este caso, puesto que la población está almacenada en un tipo de datos en forma de array con un espaciado entre elementos regular, el construir el tipo de dato derivado de la población inicial se puede construir directamente empleando la función *MPI_Type_vector()*.

Por lo que, una vez realizado por parte del nodo maestro de la lectura de los ficheros de entrada en la ejecución, y con la población inicial realiza el envío de todos los datos necesarios para comenzar con la experimentación al resto de procesadores. Como se puede observar, se encuentra dentro de este bloque una barreras de sincronización, mediante la llamada *MPI_Barrier()* que garantiza que todos los procesadores, reciben correctamente los datos antes de comenzar la ejecución en paralelo.

```
MPI_Barrier (MPI_COMM_WORLD);

/*
* Se construye el tipo derivado que representa la generacion
*/
buildTypeDerivedGeneracion (&gen_aux, &generacion);

/*
* El hilo principal envía al resto de hilos la estructura con todos los datos de la generacion
*/
MPI_Bcast(&gen_aux, 1, generacion, 0, MPI_COMM_WORLD);

/*
* Se construye el tipo derivado que contiene los parametros del experimento
*/
buildTypeDerivedParametrosExperimento(&exp_parameters_aux, &parametrosExperimento);

/*
* El hilo principal envía al resto de hilos la estructura con todos los datos del experimento
*/
MPI_Bcast(&exp_parameters_aux, 1, parametrosExperimento, 0, MPI_COMM_WORLD);

/*
* Se contruye el tipo derivado que contiene los parametros del script
*/
buildTypeDerivedScriptParameter(&script_aux, &scriptParameter);

/*
* El hilo principal envía al resto de hilos la estructura con todos los parametros del script
*/
MPI_Bcast(&script_aux, 1, scriptParameter, 0, MPI_COMM_WORLD);

/*
* Se contruye el tipo derivado que contiene la poblacion
*/
buildTypeDerivedPopulation(population, &poblacion);

/*
* El hilo principal envía al resto la poblacion inicial
*/
MPI_Bcast(&population, 1, poblacion, 0, MPI_COMM_WORLD);

/*
* El hilo principal envía nts a cada uno de los procesos
*/
MPI_Bcast(&nts,1,MPI_INT,0, MPI_COMM_WORLD);

/*
* El hilo principal envía la serie temporal normalizada
*/
        MPI_Bcast(&vector_time_serie_normal_aux,TIME_SERIE_SIZE,MPI_DOUBLE,0,MPI_COMM_WORLD);

MPI_Barrier(MPI_COMM_WORLD);
```

Fase de Agrupación

En las dos etapas anteriores se particionó el problema y se determinaron los requerimientos de comunicación. El algoritmo resultante es aún abstracto en el sentido de que no se tomó en cuenta la máquina sobre el que correrá. En esta etapa se va de lo abstracto a lo concreto y se revisa el algoritmo obtenido tratando de producir un algoritmo que corra eficientemente sobre cierta clase de computadores. En particular se considera si es útil agrupar tareas y si vale la pena replicar datos y/o cómputos.

En la fase de partición se trató de establecer el mayor número posible de tareas con la intención de explorar al máximo las oportunidades de paralelismo. Esto no necesariamente produce un algoritmo eficiente ya que el costo de comunicación puede ser significativo. En la mayoría de los computadores paralelos la comunicación es mediante el pase de mensajes y frecuentemente hay que parar los cómputos para enviar o recibir mensajes. Mediante la agrupación de tareas se puede reducir la cantidad de datos a enviar y así reducir el número de mensajes y el costo de comunicación.

La comunicación no sólo depende de la cantidad de información enviada. Cada comunicación tiene un coste fijo de arranque. Reduciendo el número de mensajes, a pesar de que se envíe la misma cantidad de información, puede ser útil. Así mismo, se puede intentar replicar cómputos y/o datos para reducir los requerimientos de comunicación. Por otro lado, también se debe considerar el coste de creación de tareas y el coste de cambio de contexto (*context switch*) en caso de que se asignen varias tareas a un mismo procesador.

En la fase de agrupación teniendo en cuenta los anteriores aspectos comentados, se consideró que la estrategia de agrupación que menor número de mensajes era necesario enviar y donde se minimiza enormemente la duplicidad de tarea en los procesadores, era asignando un número de individuos concreto de cada generación a cada uno de los procesadores.

El número de individuos se reparte proporcionalmente al número de procesadores que intervienen en la ejecución para distribuir al máximo la carga computacional entre ellos. A continuación, se puede observar el reparto efectuado:

```
int reparto (int myrank, int numeroIndividuos, int *limiteInferior, int *limiteSuperior){
  double resto = (gen_aux.population_size % numprocs);
  int desplazamiento = 0;
  if (resto == 0 || myrank >= resto){
    numeroIndividuos = (gen_aux.population_size / numprocs);
  }else{
    numeroIndividuos = (gen_aux.population_size / numprocs) + 1;
  }
  //Se calcula el desplazamiento de cada uno de los procesoss
  if (resto == 0 || myrank == 0){
    desplazamiento = 0;
  }else if (myrank >= resto){
    desplazamiento = resto;
  }else{
    desplazamiento = myrank;
  }
  //Se inicializa la variable que se encarga de almacenar el numero del primer y ultimo cuerpo tratado por cada proceso
  if (myrank == 0){
            *limiteInferior = 0;
  }else{
            *limiteInferior = myrank * (gen_aux.population_size / numprocs) + desplazamiento;
  }
  *limiteSuperior = *limiteInferior + numeroIndividuos;
  return 0;
}
```

Figura 34 - Reparto de individuos de la generación entre los procesadores de la ejecución

4. Implementación Paralelización OpenMP

En el presente capítulo se pretende tratar en profundidad la técnica de implementación seguida de la paralelización del sistema empleando la librería de OpenMP basada en la memoria compartida.

Antes de poder realizar la paralelización del sistema, se tuvo que analizar exhaustivamente el código de la versión secuencial para comprender perfectamente el funcionamiento de éste y así poder identificar aquellas zonas de código susceptibles de paralelizarse. A continuación se detalla el razonamiento seguido para definir la estrategia de paralelización.

Descomposición del problema

Esta primera etapa se centró en la definición de pequeñas tareas que conformasen el algoritmo con el fin de convertirlo en grano fino. Al igual que en planteamiento de la paralelización de MPI, a la hora de estudiar dicha descomposición, se tuvo en cuenta que la estrategia de paralelización a seguir debía reducir al máximo la duplicación de datos y computación.

Tal y como se realizó la descomposición del problema en MPI, se ha empleado la técnica del algoritmo maestro-esclavo. Se trabaja con una única población de individuos que será gestionada por el nodo maestro. La evaluación de adecuación de los individuos y/o la aplicación de operadores genéticos puede ser aplicada por los nodos esclavos. A cada nodo esclavo le corresponderá una parte de la población total, sobre la cual realizará las operaciones antes citadas. Una vez terminado este proceso, devolverán el resultado al nodo maestro, que realizará la selección de individuos.

Aplicados los puntos anteriores, a continuación se detalla la descomposición del problema.

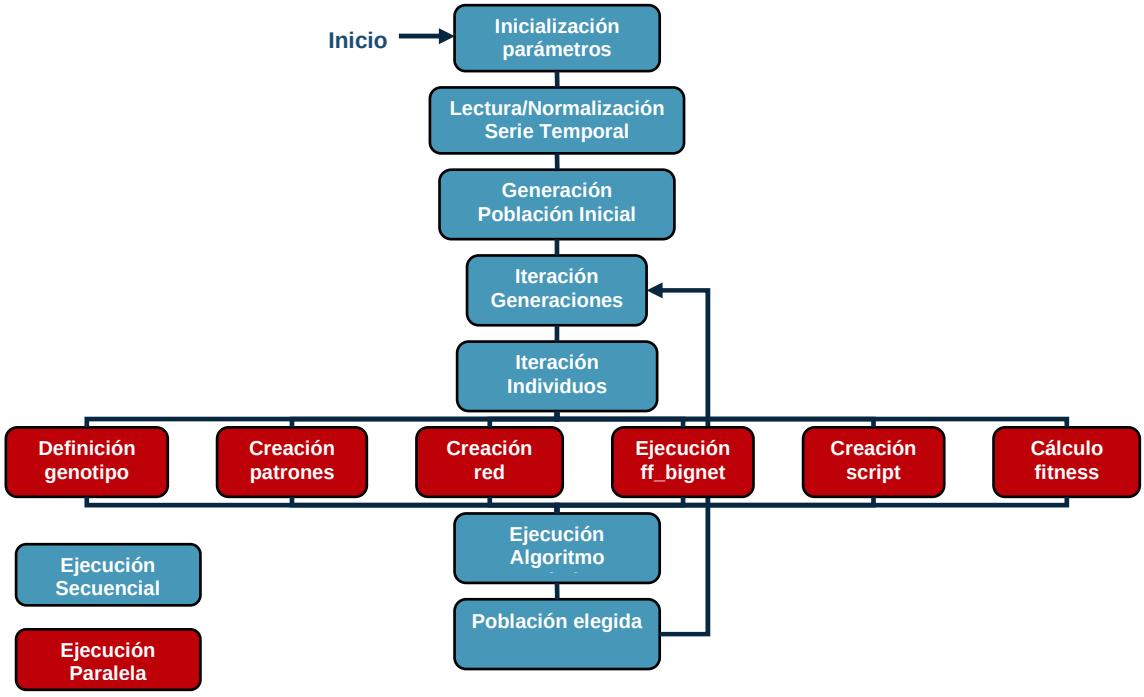

Figura 35 - Diseño de la descomposición de la paralelización del algoritmo

La operación que se implementa en paralelo es la evaluación de la adecuación de los individuos, porque suele ser la más compleja. Además, este valor es independiente del resto de la población, por lo que su implementación es sencilla.

Fase de comunicación

La comunicación con la descomposición anteriormente comentada se basa en que en función de la política de reparto de OpenMP (*dynamic, static, runtime, …)* se reparte la población de individuos de cada generación a cada uno de los nodos esclavos y éstos le devuelven los valores de adecuación de todos los individuos para generar la siguiente generación.

La comunicación empleada es síncrona puesto que el nodo maestro espera a que todos los nodos esclavos envíen sus valores de adecuación de todos los individuos para generar la siguiente generación. Se podría haber implementado mediante una comunicación asíncrona, es decir, el algoritmo no espera a los nodos más lentos a que envíen sus *fitness*. De este modo agilizaríamos el proceso, pero el comportamiento no sería exactamente el mismo que el de un algoritmo genético secuencial. La implementación síncrona, sin embargo, si mantiene este comportamiento.

Fase de Agrupación

En la fase de agrupación es OpenMP quien de acuerdo a la directiva *schedule* junto con el parámetro *chunk* determina cuál va a ser el reparto de individuos de la población entre cada uno de los nodos esclavos.

Dichos parámetros se permiten determinar en el momento de ejecución. El criterio de reparto es el siguiente:

- ***static***: las iteraciones se dividen en *chunks* de un tamaño definido.

- ***dynamic***: cada *thread* ejecuta un *chunk* de iteraciones y al terminar su parte busca otro *chunk* para procesar.

- ***guided***: cada *thread* ejecuta un *chunk* de iteraciones y al terminar su parte busca otro *chunk* para procesar. El tamaño de *chunk* varía, siendo grande al inicial y éste se va reduciendo.

- ***auto***: la decisión del tipo de schedule es decidido por el compilador/sistema operativo.

- ***runtime***: el schedule y el tamaño de *chunk* son tomados de la variable *sched-var ICV*.

La ventaja de haber realizado la paralelización mediante nodos maestro-esclavo tal y como se ha detallado anteriormente, es que, sea cual sea la política de agrupación siempre existirá los mismos costos de comunicación, puesto que la cantidad de mensajes que envían los nodos esclavos al nodo maestro no varía en función de la agrupación.

Además, es una estrategia de paralelización que asegura que las tareas resultantes de la agrupación tienen costos de computación y comunicación muy similares y que en el caso de ampliar el número de procesadores disponibles, será un algoritmo con paralelización potencial para cualquier número de procesadores.

A la hora de realizar la paralelización, se ha dejado la elección de la política de planificación al usuario de forma que a la hora de ejecutar el programa realizará la llamada de acuerdo a:

./ag -P (nombre fichero parametros) -t (nombre fichero serie temporal) -n (0) -np (numero de procesadores) -pl (STATIC, DYNAMIC, RUNTIME, GUIDED) -c (numero de granularidad)

Como se puede observar, los dos últimos parámetros en la ejecución, son parámetros referentes a la agrupación que posteriormente se realizará de individuos para cada uno de los procesadores.

A la hora de realizar la paralelización con OpenMP, se debe cambiar el paradigma a la paralelización mediante memoria compartida, por lo que, lo más crítico una vez identificado los posibles puntos de paralelización, es identificar aquellas variables que se deben compartir por parte de todos los procesadores y cuáles se pueden considerar privadas.

Además, teniendo en cuenta que antes y después de las zonas paralelas, definidas con la directiva #*pragma parallel*, las variables no tienen valores definidos, era necesario determinar qué variables entran definidas con valores

que es necesario mantener en la zona paralela y qué variables deben mantenerse con los valores de la paralelización tras finalizar la zona paralela. De tal forma que, a la hora de realizar la paralelización de las poblaciones del algoritmo, se realizó mediante la siguiente directiva:

#pragma omp parallel for private (i,identificador, individuo,int_input, int_hidden, final_path, genotipo, learning, netname, result, script, matrix_patterns_aux, vector_kind_patterns_aux) firstprivate (script_aux)

La presente directiva, define como privadas aquellas variables en las que es necesario proteger su cambio. De forma que, cada thread tiene una copia local de las variables. La variables definidas como privadas son invisibles para los demás *threads*. Además, no están definidas ni al entrar ni al salir.

Las variables definidas como públicas son aquellas que son de sólo lectura accesibles por diferentes *threads*, como pueden ser las variables donde se almacenan los parámetros del experimento. O aquellas variables, que acceden a localizaciones como puede ser el caso de la variables *fitness* o aquellas sobre las que necesitamos comunicar valores entre diferentes *threads*. Todas aquellas variables que no se definen específicamente como privadas, por el compilador son consideradas como públicas.

Por último, como se puede observar la variable *script_aux*, se ha definido adicionalmente como *firstprivate* lo cual, cataloga la variable como privada realiza una copia para cada *thread* de acuerdo a los valores que tiene justo antes de entrar en la zona paralela.

5. Experimentación

En el siguiente apartado, se detalla la experimentación llevada a cabo tanto para la implementación mediante paso de mensajes, MPI, como para la implementación mediante memoria compartida, OpenMP.

En primer lugar, se detallará, las series temporales empleadas para llevar a cabo la implementación, y posteriormente, se detallarán los resultados obtenidos tanto como para MPI y OpenMP, para posteriormente poder realizar un análisis de los resultados de la experimentación.

Para llevar a cabo la experimentación, se han usado cinco series temporales. Estas series temporales son concretamente Abraham12, Dow-Jones, Passengers, Temperature, Paper. La serie temporal Passengers representa el número de pasajeros de una aerolínea internacional medidos en miles y mensualmente desde Enero de 1949 hasta Mayo de 1959, la fuente es de Box & Jenkins (1976). La serie temporal Temperature muestra la temperatura media mensual del aire del castillo de Nottingham medida desde 1920 hasta 1938, en este caso la fuente es de O.D. Anderson (1976). Dow-Jones trata sobre el valor mensual de cierre del índice industrial dow-jones desde Agosto de 1969 hasta Abril de 1979, la fuente es de Hipel y Mcleod (1994). La serie temporal Abraham12 representa la demanda mensual de gasolina en Ontario, en millones de galones, desde 1960 hasta 1975. La serie temporal Paper son las ventas mensuales de papel escrito o impreso en Francia desde 1963 hasta 1972. La fuente es O'Donovan (1983).

A continuación, mostraremos una gráfica de cada una de las series temporales propuestas. Todas las series temporales seleccionadas son series temporales que provienen del mundo real.

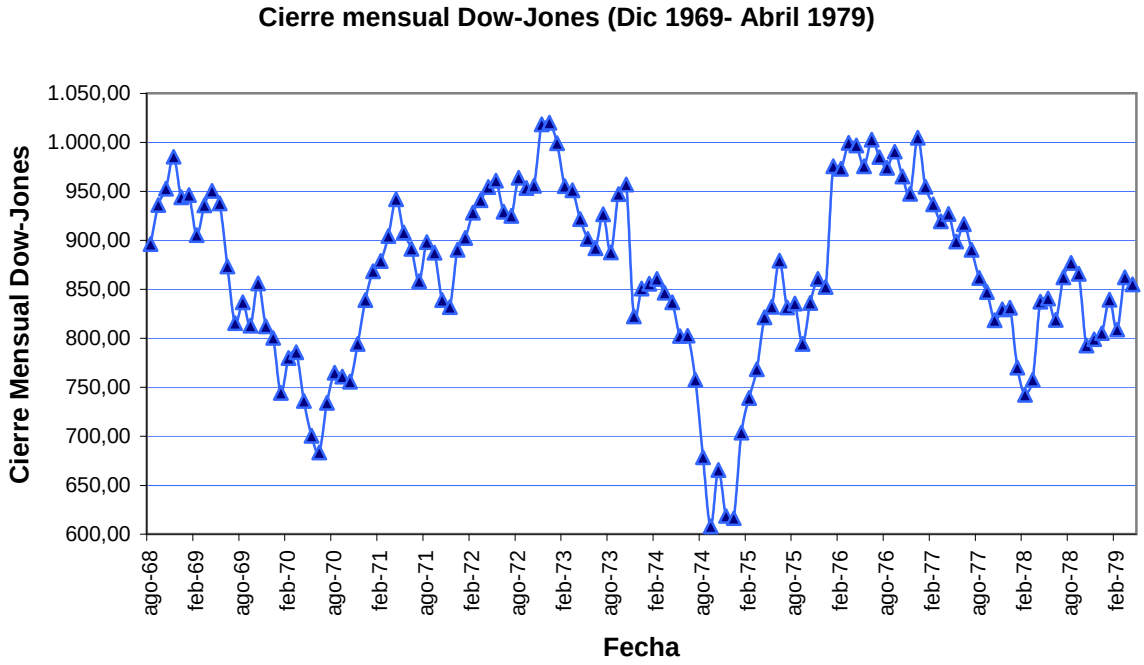

Figura 36 - Serie Temporal Dow-Jones

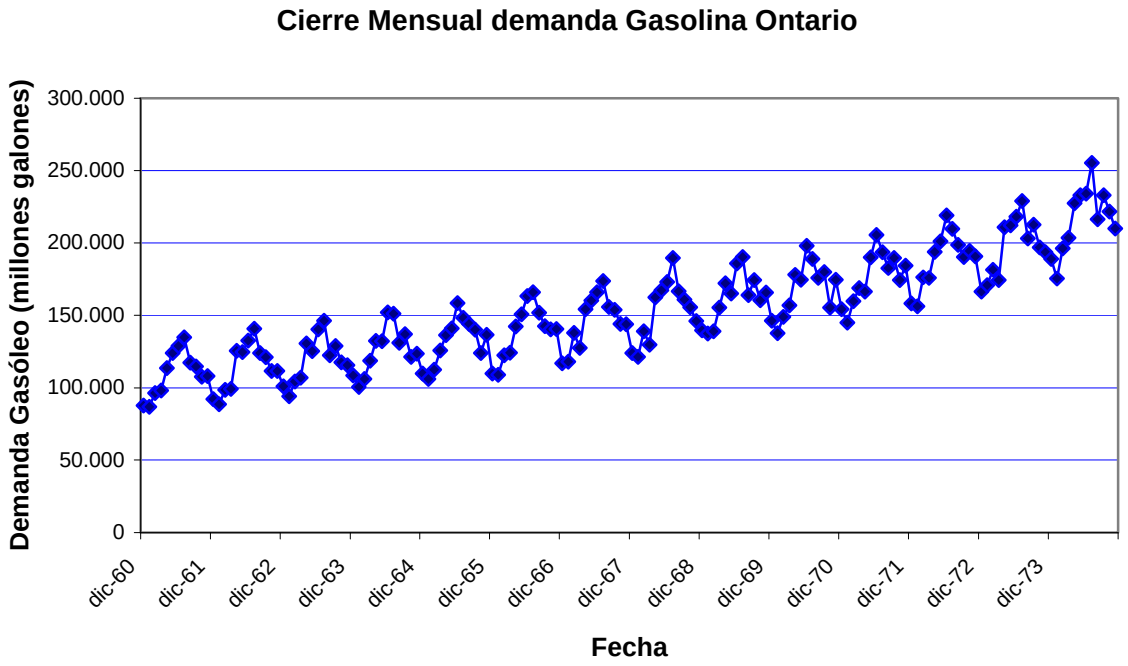

Figura 37 - Serie Temporal Abraham12

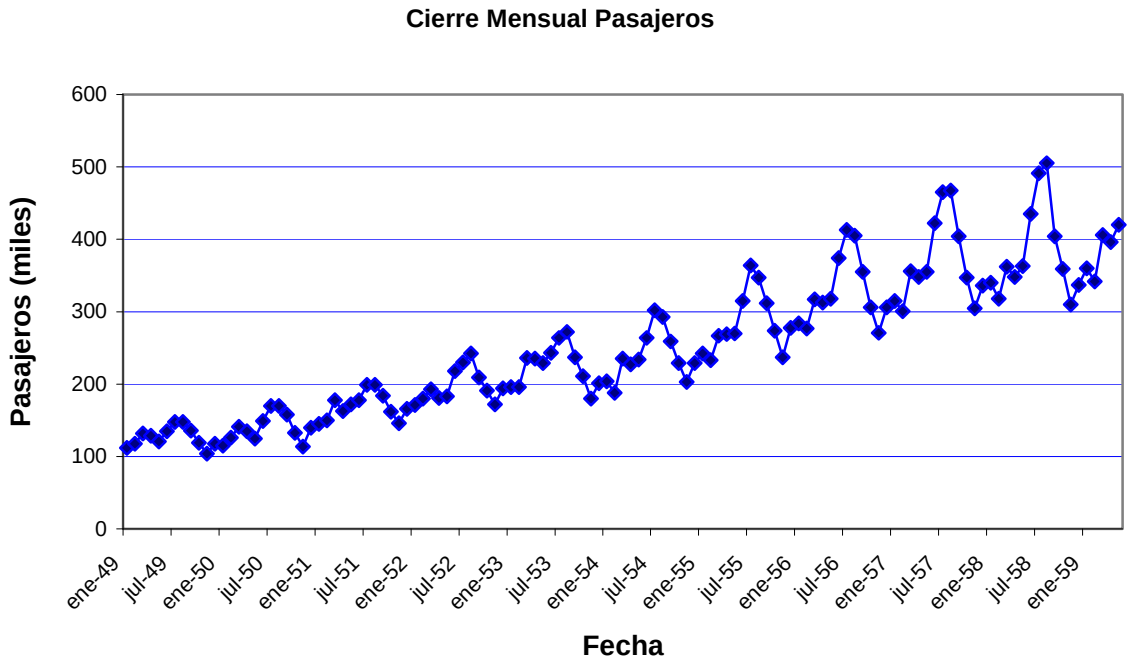

Figura 38 - Serie Temporal Passengers

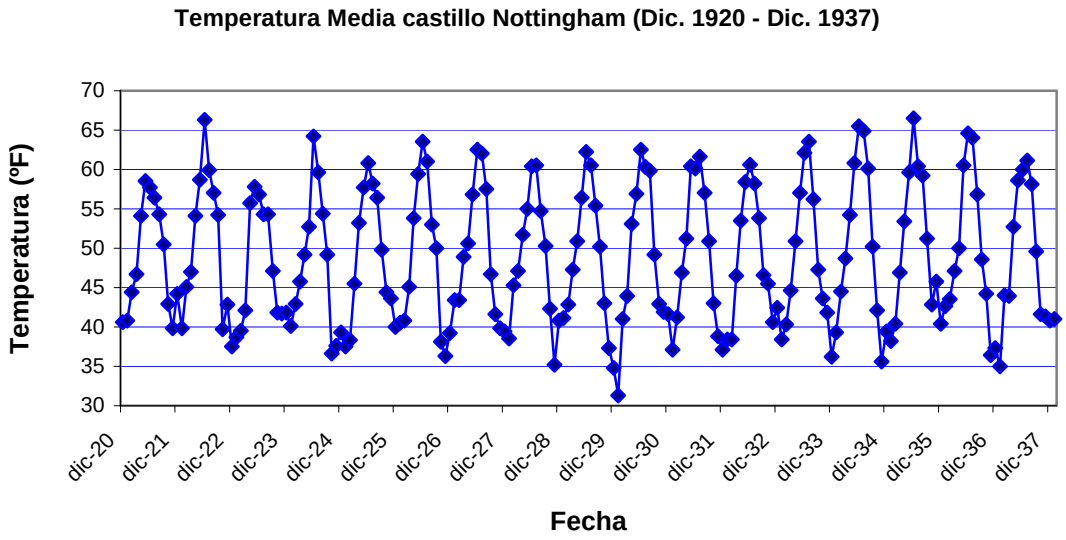

Figura 39 - Serie Temporal Temperature

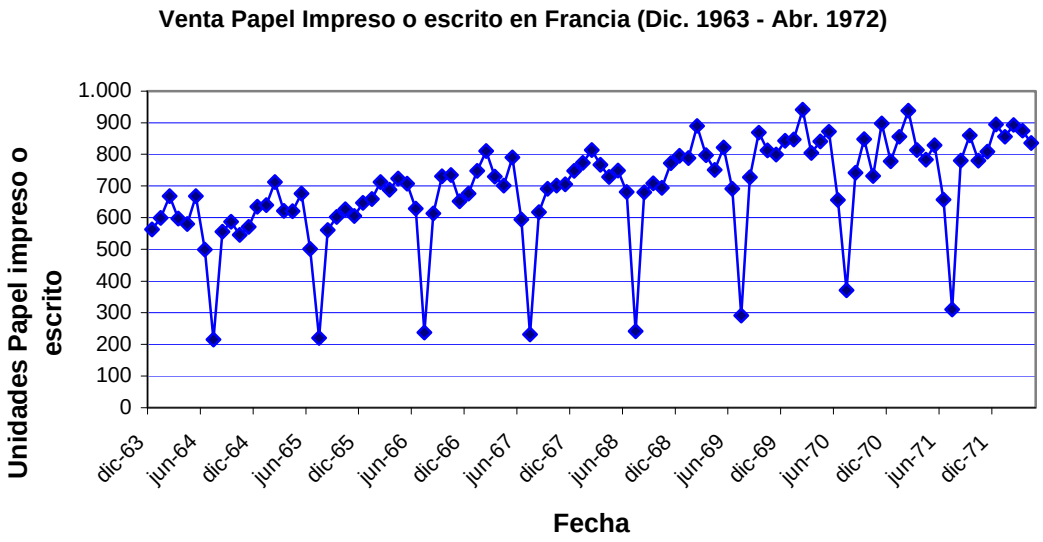

Figura 40 - Serie Temporal Paper

Todas las series temporales seleccionadas, provienen del mundo real. Los datos obtenidos del mundo real pueden sufrir por cuestiones externas tales como tormentas, terremotos, inundaciones, huelgas, guerras, avances tecnológicos, etc., lo que las hace más interesantes y difíciles de predecir.

En el siguiente cuadro, se muestra el tamaño de las series temporales, ya que puede ser un factor a tener en cuenta posteriormente, en el análisis de la experimentación:

Serie Temporal	Tamaño
Paper	101
Passengers	125
Dow-Jones	129
Abraham12	168
Temperature	206

Tabla 12 - Tamaño de las series temporales

La experimentación llevada a cabo ha consistido en realizar distintas pruebas tomando los tiempos para cada una de ellas, empleando la implementación tanto con el el estándar de MPI como con el estándar OpenMP. Para cada uno de los estándares se han realizado las pruebas para cada serie temporal con distinto número de procesadores (desde 1 procesador, ejecución secuencial, hasta 6 procesadores).

La experimentación se ha ejecutado en un servidor que ha puesto a disposición la Universidad Carlos III de Madrid con las siguientes características:

- **Servidor HP ProLiant serie ML350 G6**
- **Procesador**: Dos procesadores Intel® Xeon® E5504 (4 núcleos, 2,00 GHz, 4 MB L3, 80W)
- **Unidades de disco duro incluidas**: SAS formato pequeño; 146 GB
- **Memoria**: 16 Gb
- **Controlador de almacenamiento**: Smart Array P411/ 256 MB BBWC

El acceso de ejecución de programas en dicho servidor se ha restringido mientras se realizaban las pruebas correspondientes al proyecto, por lo que, los tiempos obtenidos en cada una de las pruebas se han obtenido bajo un mismo entorno de ejecución, lo cual, permite obtener conclusiones con mayor fiabilidad.

5.1 Experimentación con MPI

El primero de los resultados de la experimentación que se va a mostrar se trata del tiempo de ejecución. A la hora de realizar la medición del tiempo de ejecución de la aplicación paralela, se analizaron distintas alternativas posibles. Para cada una de las alternativas se tuvieron en cuenta los siguientes factores involucrados en los criterios de medición de tiempo:

1. **Resolución**: lo que representa la precisión del sistema de medición. Por ejemplo, el reloj de software (clock) en un sistema operativo puede tener resolución de 0,001 segundos.

2. **Exactitud**: representa la cercanía del valor medido al verdadero. Se puede cuantificar en términos del error de medición.

3. **Granularidad**: es la unidad software que se quiere medir: programa (granularidad gruesa), función, instrucción (granularidad fina), etc.

4. **Complejidad**: representa cuán difícil es obtener la medición.

Se analizaron distintas alternativas de técnicas de medición, y para tomar la decisión de cuál se consideraba más adecuada, se cuantificaron los cuatro criterios comentados anteriormente, llegando a:

Método	Resolución	Exactitud	Granularidad	Complejidad
Reloj de pulsera	0,01 s	0,5 s	proceso	fácil
Comando date	0.02 s	0.02 s	proceso	fácil
Comando time	0.02 s	0.02 s	proceso	fácil
prof y gprof	10 ms	20 ms	funciones	moderada
clock()	10-30 ms	15-30 ms	instrucción	moderada
Contadores del procesador	0,4-4 µs	1-8 µs	instrucción	muy difícil

Tabla 13 - Análisis de técnicas de medición

El reloj de pulsera de primeras se consideró un método con una exactitud poco asumible y que a pesar de tratarse de un método sencillo, el tamaño del problema a resolver, era tan grande que obligaba permanecer delante de la monitorización de la ejecución desde el inicio al fin de la ejecución, por lo que, fue un método desechado desde el inicio.

El comando *date*, presentaba una exactitud asumible, y ofrecía la ventaja de no necesitar permanecer delante de la monitorización de la ejecución. Pero al igual que el reloj de pulsera, ofrece el tiempo total considerando tiempos de procesador, tiempos de espera, tiempos I/O, etc.

El comando *time*, presentaba las mismas características que el comando date, pero con la gran ventaja de poder obtener tiempos de ejecución en modo usuario y modo kernel. Se trata de la opción elegida, debido a que su medición es sencilla y presenta una exactitud considerada adecuada para la experimentación que se ha realizado.

El comando *prof* y *gprof*, presentan mayor precisión y se ha empleado también en la experimentación, pero no con la misión de informar del tiempo de ejecución ya que la exactitud obtenida con el comando time se considera suficiente, sino que, son comandos que ofrecen información adicional de la ejecución que se detallará más adelante en la sección de Trabajos Futuros que son interesantes para poder analizar las posibilidades de mejorar en la paralelización empleada.

El comando clock(), además de tratarse de un método de medición más complejo, no nos permite distinguir entre el tiempo de usuario y de sistema, por lo que, pese a mejor exactitud, no ofrecía una mejora respecto a los comandos *time* y *gprof* empleados.

Los contadores del procesador, se puede considerar como el método más “purista” pero a la hora de realizar la medición del tiempo de ejecución, se trata de un método muy complejo para medir, y necesitar una política de monitorización muy elaborada para poder llegar a separar el tiempo de usuario y tiempo de sistema.

Para realizar la medición del tiempo de ejecución, por lo tanto, se ha empleado el comando linux *.time*. El comando *.time* en GNU/Linux nos permite conocer muchos detalles sobre el tiempo de ejecución de otro comando o aplicación que se ejecute. Cuando se ejecuta el comando *.time* se obtiene una salida similar a la siguiente:

Donde ***real*** indica el tiempo que ha pasado desde que se ejecuta el comando hasta que termina. ***User*** indica el tiempo usado por el proceso en modo usuario y ***sys*** muestra el tiempo usado por el sistema en manejar el proceso (modo kernel).

Pero a la hora de emplear dicho comando para la medición de programas paralelos es necesario tener en cuenta los siguientes puntos:

- *Real* indica el tiempo que ha pasado desde que se ejecuta el programa o comando hasta que el último de los *threads* o procesos finaliza la ejecución.

- *User* indica el tiempo acumulado en la ejecución de cada unos de los *threads* o procesos.

A continuación, se muestra una gráfica en donde se reflejan los tiempos obtenidos para cada una de las series temporales y el número de procesadores empleados de la siguiente tabla:

#procesadores	Paper	Passengers	Dow-Jones	Abraham12	Temperature
1	16.594,64	28.506,78	26.001,83	34.297,52	55.582,04
2	13.442,47	19.211,39	21.015,81	24.977,65	46.143,62
3	10.149,00	16.826,84	11.301,24	22.656,25	25.478,07
4	9.734,47	13.559,48	11.977,87	18.560,53	21.884,49
5	**8.183,67**	12.177,58	**10.138,44**	**13.806,02**	**15.748,41**
6	10.376,34	**11.054,15**	10.756,16	14.440,66	22.049,88

Tabla 14 - Comportamiento tiempo ejecución MPI

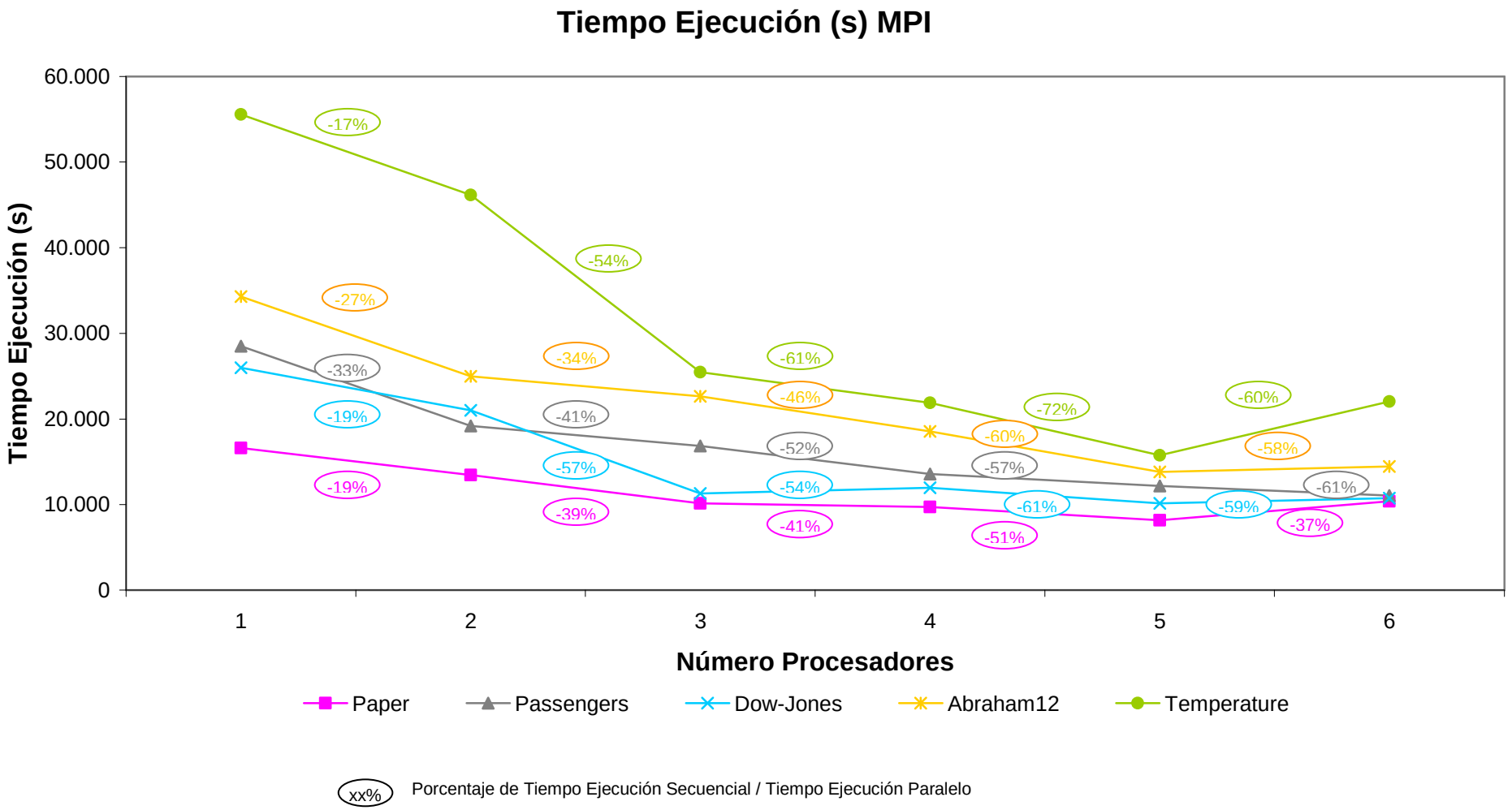

Figura 35 Comportamiento Tiempo Ejecución MPI

Como se puede observar en la gráfica, con la paralelización implementada mediante el paso de mensajes, MPI, se obtienen reducciones de tiempo de ejecución considerables. Cuanto mayor es el número de procesadores, mayor es la reducción de tiempo llegando a alcanzar hasta una reducción del 72% para la serie temporal Temperature empleando cinco procesadores.

Además, en todas las series temporales, la reducción máximo respecto al tiempo de ejecución secuencial se logra en los cinco procesadores, salvo para la serie temporal Passengers que lo alcanza con seis procesadores. Pero a partir de los cinco procesadores, se observa un pequeño aumento en el tiempo. Lo cual iría en contra de toda lógica, que haría pensar que cuanto mayor número de procesadores, mayor mejora en tiempo se obtendría.

Este comportamiento es común dentro de los sistemas paralelos, y se reflejó dentro de la teoría de sistemas paralelos en la Ley de Amdahl. La Ley de Amdahl menciona que un programa paralelo tiene una parte secuencial que eventualmente limita la aceleración que se puede alcanzar en una plataforma paralela. A continuación se exponen razones básicas por las que P procesadores pueden no aumentar la rapidez de cómputo por un factor P y son:

1. ***Overhead***: se denomina *overhead* a cualquier costo que se incurre en la solución paralela pero no en la secuencial. Este coste se debe a:
    a. **Coste de creación de *threads* o procesos**: el *kernel* del sistema operativo, siempre a la hora de crear un *thread* o proceso necesita un tiempo para la creación de las estructuras de datos, y reservas de memoria necesarias para la gestión de procesos.
    b. **Coste de comunicación entre procesos o *threads***: la comunicación en los *threads* o procesos es una gran componente de *overhead* dado que un cómputo secuencial no tiene que comunicarse con otro procesador, toda la comunicación es una forma de *overhead*.
    c. **Coste de sincronización**: coste de *overhead* que aparece cuando los *threads* o procesos deben esperar por un evento en otro *thread* o proceso.

2. **Computación no paralelizable**: en la paralelización de algoritmos, es importante considerar que todo programa paralelo tiene una parte secuencial que eventualmente limita la aceleración que se puede alcanzar en una plataforma paralela (Ley de Amdahl). Estos fragmentos de código siempre se ejecutarán por el proceso maestro y en muchos casos implica que el resto de procesadores tengan que realizar una espera de la ejecución del fragmento no paralelizable.

3. **Procesadores inactivos**: a la hora de realizar la paralelización de un algoritmo, en función del balance de la carga que se realice entre los procesadores, existirá procesadores que finalicen su cómputo antes que el resto, y por tanto, deberán esperar a la finalización de estos (coste de sincronización) para poder continuar con la ejecución.

4. **Contención de recursos**: en los sistemas existen recursos que en muchos casos requieren un acceso secuencial, es decir, no es posible que se acceda a ellos desde distintos procesadores de forma simultánea, lo cual, puede llegar a ser un cuello de botella a la hora de realizar la paralelización de algoritmos.

Un comportamiento que además se puede observar en la gráfica de los tiempos de ejecución, es que, la mejora respecto al tiempo de ejecución secuencial es mayor cuanto mayor tamaño tiene la serie temporal. Lo cual tiene sentido ya que cuanto mayor es el tamaño de la serie temporal, mayor cómputo debe realizar el algoritmo secuencial, a la hora de paralelizar el algoritmo, este cómputo se reparte entre los procesadores que intervienen en la paralelización, y por tanto, el ahorro en tiempo de ejecución respecto a la ejecución secuencial será mayor.

Dentro de la teoría de sistemas paralelos, la aceleración del tiempo de ejecución secuencial respecto al paralelo se conoce como *speed-up*. Existen dos definiciones de aceleración que varían en función de qué se emplea como tiempo de ejecución secuencial.

1. **Aceleración relativa**: el tiempo de ejecución secuencial usado es el tiempo de ejecución del programa paralelo cuando éste es ejecutado sobre un único procesador del computador paralelo.

2. **Aceleración real**: el tiempo de ejecución secuencial usado es el tiempo del mejor programa secuencial para resolver el problema.

Con las anteriores definiciones, se ha optado por tomar la Aceleración real como medida ya que se ha considerado que a la hora de realizar comparaciones, nos da una visión más realista. Más realista, puesto que al haberse realizado la implementación del algoritmo paralelo con MPI y OpenMP, éstos paradigmas no emplean la misma técnica de reparto de cargas, en el caso de MPI el reparto se debe incluir dentro del código de paralelización, mientras que en el caso de OpenMP, el compilador es el encargado de realizar dicho reparto en función de una serie de directivas que permite al programador determinar el criterio de reparto.

Por lo que, considerando la aceleración, o *speed-up*, como la medida que expresa el beneficio obtenido en tiempo cuando usamos paralelismo y cuál es la aceleración que resulta por dicho paralelismo.

$$S(n) = T(1)/T(n)$$

Se representa en el siguiente gráfico su comportamiento para cada una de las series temporales empleadas en la experimentación.

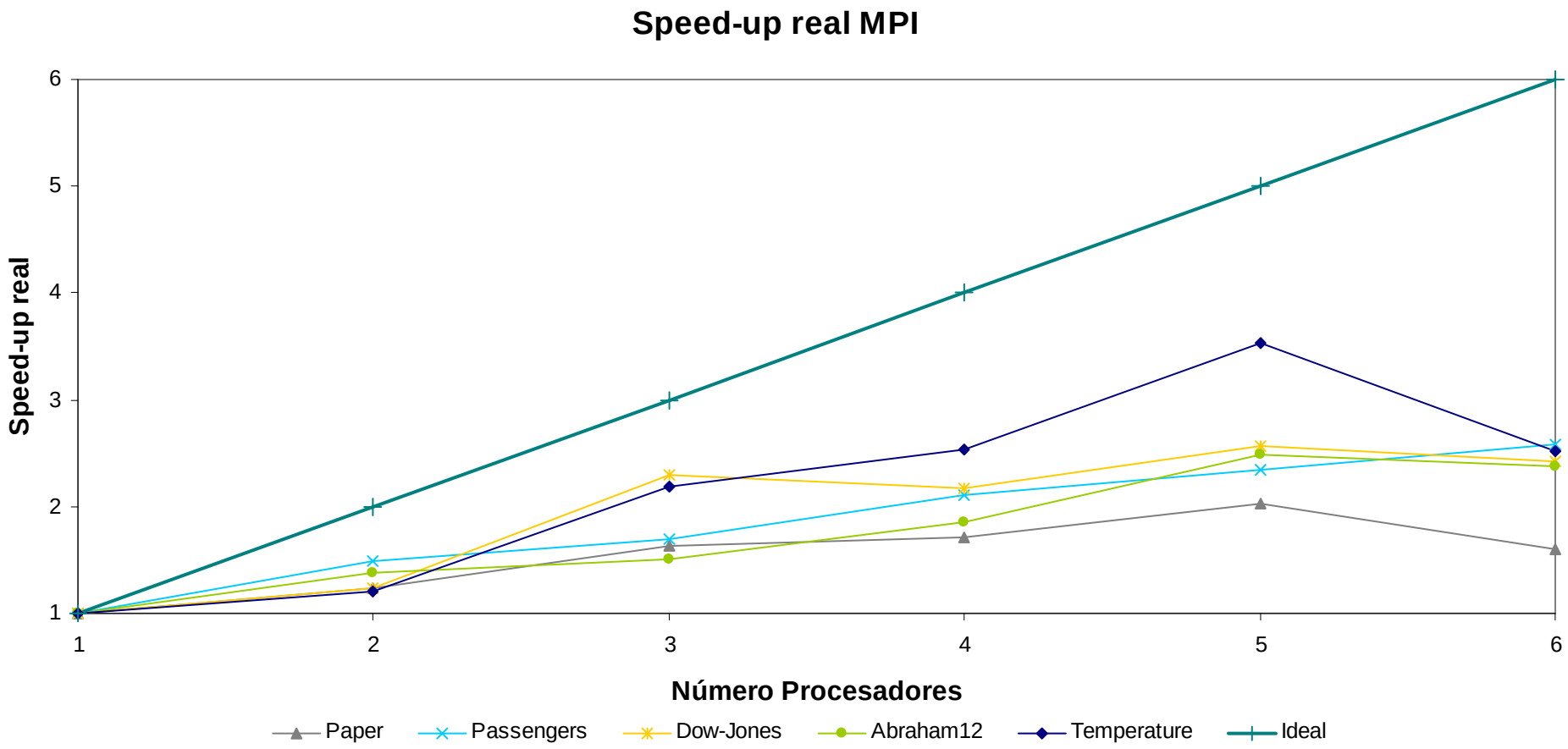

Figura 36 Comportamiento *Speed-Up* MPI

A continuación, se muestran los valores de la gráfica, para poder analizarlos:

#procesadores	Paper	Passengers	Dow-Jones	Abraham12	Temperature
1	1,00	1,00	1,00	1,00	1,00
2	1,23	1,48	1,24	1,37	1,20
3	1,64	1,69	2,30	1,51	2,18
4	1,70	2,10	2,17	1,85	2,54
5	**2,03**	2,34	**2,56**	**2,48**	**3,53**
6	1,60	**2,58**	2,42	2,38	2,52

Tabla 15 - Comportamiento speed-up MPI

Tal y como ya se ha comentado anteriormente, los mejores resultados en tiempo se obtienen en prácticamente todas las series temporales con 5 procesadores. A partir de los cinco procesadores se puede observar, un ligero aumento en el tiempo de ejecución.

Pese a que el objetivo del proyecto es reducir al máximo el tiempo de ejecución, también es importante analizar el comportamiento de la paralelización implementada respecto a el aprovechamiento de los recursos disponibles, procesadores. Para ello, es interesante mostrar el comportamiento de la eficiencia en función de cómo se vaya incrementando el número de procesadores.

En muchas ocasiones, un algoritmo secuencial es evaluado en términos de su tiempo de ejecución, expresado como una función del tamaño de su entrada. El tiempo de ejecución de un sistema paralelo depende no sólo del tamaño de su entrada sino también de la arquitectura paralela, el número de procesadores y característica de la máquina tales como: velocidad del procesador, velocidad de los canales de comunicación, topología de interconexión y técnicas de ruteo. Por este motivo no puede evaluarse un algoritmo paralelo aisladamente de la arquitectura paralela sobre la que es implementado.

Un algoritmo que posee una buena performance para un problema seleccionado o sobre un número determinado de procesadores en una máquina dada puede funcionar pobremente si alguno de los parámetros cambia. La escalabilidad de un algoritmo paralelo sobre una arquitectura es una medida de su habilidad de obtener *speed-up* creciente linealmente con respecto al número de procesadores, en consecuencia refleja la capacidad del sistema paralelo para usar efectivamente una cantidad creciente de recursos de procesamiento.

Considerando la eficiencia como el porcentaje de tiempo empleado en proceso efectivo, representado por la ecuación:

$$E(n) = \frac{S(n)}{n} = \frac{T(1)}{nT(n)}$$

Los datos representados en la gráfica se muestran a continuación:

#procesadores	Paper	Passengers	Dow-Jones	Abraham12	Temperature
1	100,00	100,00	100,00	100,00	100,00
2	61,72	74,19	61,86	68,66	60,23
3	54,50	56,47	76,69	50,46	72,72
4	42,62	52,56	54,27	46,20	63,49
5	40,56	46,82	51,29	49,68	70,59
6	26,65	42,98	40,29	39,58	42,01

Tabla 16 - Comportamiento eficiencia MPI

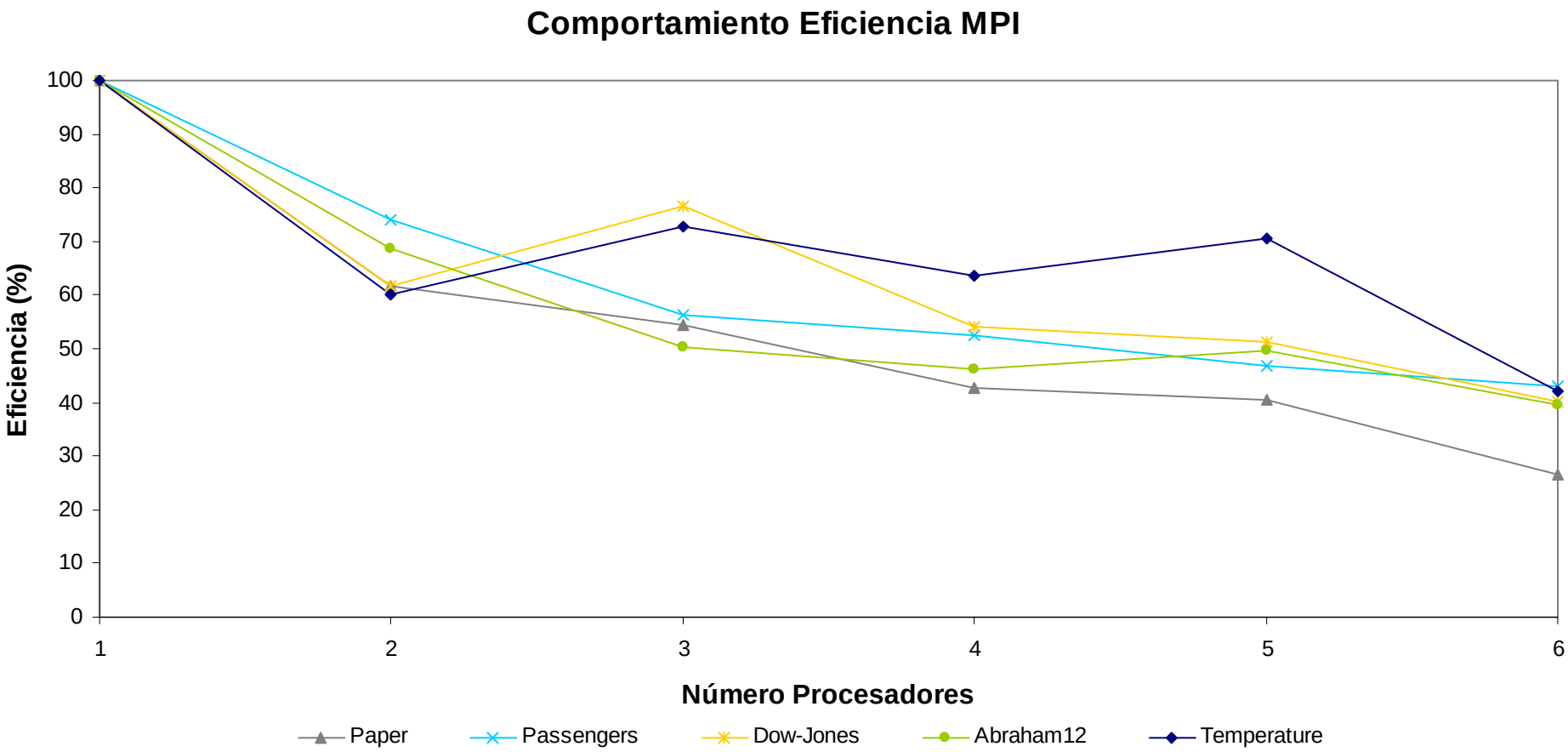

Figura 37 Comportamiento eficiencia MPI

Como se puede observar en la gráfica, la eficiencia el sistema paralelo decrece cuando se incrementa el número de procesadores. Lo cual se trata de algo lógico, puesto que los costes de *overhead*, sincronización incrementan cuando se incrementa el número de procesadores. Sin embargo, para una instancia más grande del mismo problema se tiene mejor *speed-up* y eficiencia para el mismo número de procesadores.

En los sistemas paralelos no se puede mantener las prestaciones considerando un tamaño fijo del problema y aumentando el número de procesadores. Se debe buscar el mantener las prestaciones al aumentar el tamaño del sistema, pero aumentando a su vez el tamaño del problema.

A la hora de analizar los sistemas paralelos, la idea es ser capaces de determinar si en un futuro, cuando aumente el número de procesadores y el tamaño de los problemas que se quieren resolver, se seguirán manteniendo las prestaciones. Para ello surge la función de isoeficiencia, que pretende dar idea de cómo debe crecer el tamaño del problema en función del número de procesadores para mantener la eficiencia del sistema constante.

5.2 Experimentación con OpenMP

Al igual que en la experimentación de MPI la primera de las medidas tomadas para analizar el comportamiento del algoritmo implementado con OpenMP, es el tiempo de ejecución.

Para la medición del tiempo de ejecución, se ha empleado el comando *.time* al igual que en la experimentación de MPI. Para la experimentación de OpenMP se ha empleado el mismo servidor, de forma, que los tiempos obtenidos podrán ser comparados con posterioridad para llegar a posibles conclusiones sobre la implementación más adecuada.

Es importante mencionar que OpenMP permite incluir dentro de sus directivas existe la capacidad de detallar el modo de distribuir las iteraciones de los bucles entre los *threads*. Las distintas opciones de esta directiva son:

- *SCHEDULE (STATIC, chunk):* distribuye un subconjunto de iteraciones en cada thread de forma circular. El tamaño del subconjunto viene dado por chunk. Si chunk no viene informado, la distribución es por bloques.

- *SCHEDULE (DYNAMIC, chunk):* igual que el anterior pero ahora los bloques son repartidos dinámicamente a medida que los *threads* van finalizando su ejecución.

- *SCHEDULE (GUIDED, chunk):* con N iteraciones y P *threads* disminuye dinámicamente N/kP iteraciones a cada *thread* (donde k es una constante que depende de la implementación). El valor de N se va actualizando con el valor de las iteraciones que quedan sin asignar hasta un valor mínimo dado por chunk. La asignación se hace por orden de finalización.

- *SCHEDULE (RUNTIME):* el tipo de planificación se define en tiempo de ejecución mediante la variable de entorno (*OMP_SCHEDULE*).

La implementación llevada a cabo, permite al usuario definir a la hora de ejecutar el programa paralelo, qué política de reparto emplear. Pero a la hora de realizar la experimentación, siempre se ha empleado la misma política, *SCHEDULE(DYNAMIC,1)*.

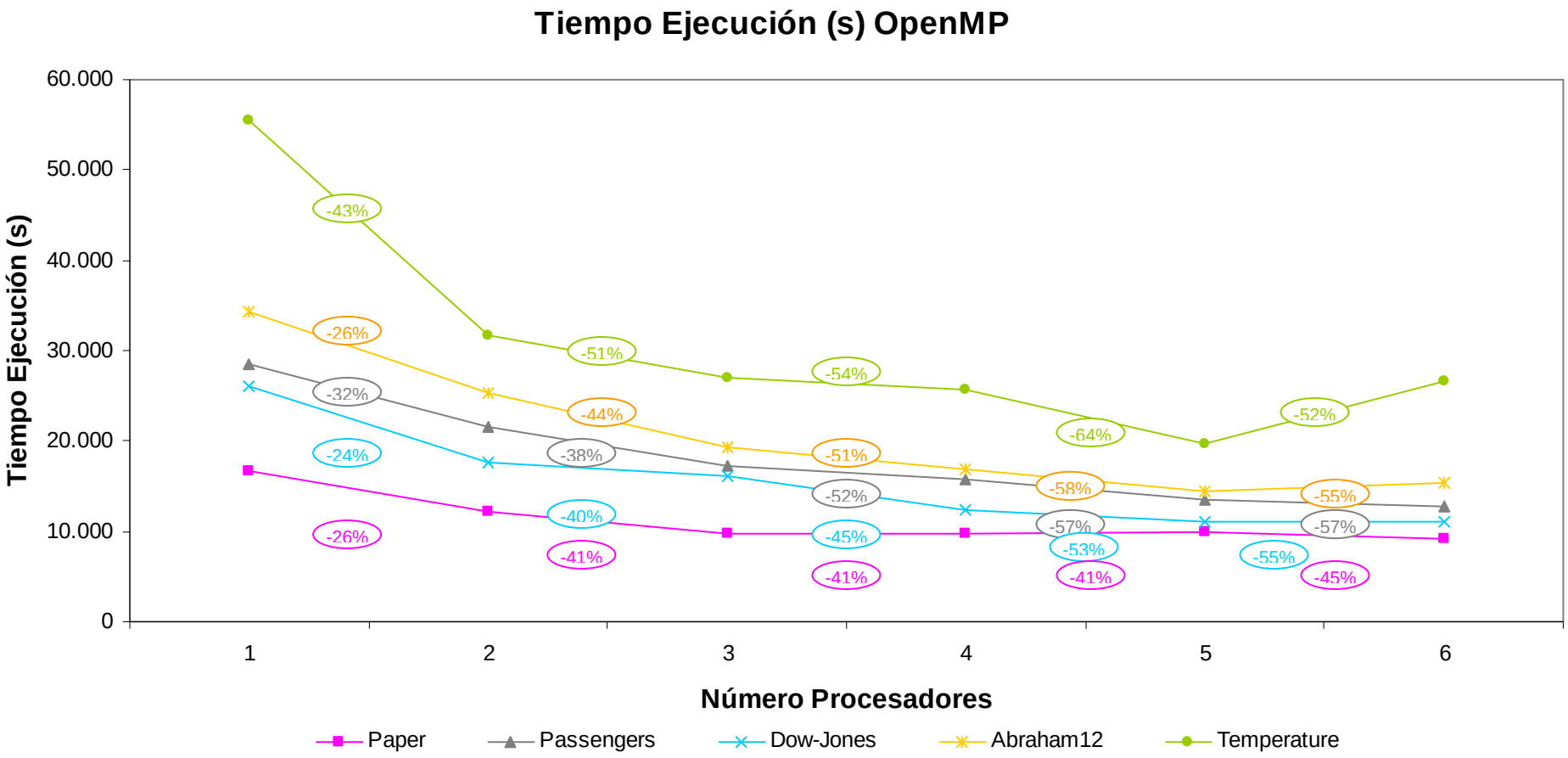

Figura 38 Comportamiento Tiempo Ejecución OpenMP

A continuación, para poder analizar los tiempos de ejecución, se muestran los tiempos representados en la gráfica:

#procesadores	Paper	Passengers	Dow-Jones	Abraham12	Temperature
1	16.594,64	28.506,78	26.001,83	34.297,52	55.582,04
2	12.228,18	21.581,92	17.630,27	25.336,81	31.759,01
3	9.751,35	17.181,38	16.134,98	19.337,18	27.090,26
4	9.792,42	15.796,87	12.465,64	16.919,31	25.717,15
5	9.866,26	13.426,04	11.068,11	**14.452,57**	**19.742,84**
6	**9.116,08**	**12.814,13**	**11.062,90**	15.335,49	26.672,60

Tabla 17 - Comportamiento tiempo ejecución OpenMP

Como se puede observar tanto en la gráfica como en la tabla de tiempos de ejecución, las series temporales de menor tamaño (Paper, Passengers, Dow-Jones) presentan la mayor reducción de tiempo de ejecución en seis procesadores. Las series temporales con mayor tamaño (Abraham12 y Temperature) al incluir un sexto procesador aumentan su tiempo de ejecución.

Al igual que en el análisis del tiempo de ejecución de MPI, se puede observar que P procesadores no pueden aumentar la rapidez de cómputo por un factor P. Este comportamiento es común dentro de los sistemas paralelos, y se reflejó dentro de la teoría de sistemas paralelos en la Ley de Amdahl. Unos de los motivos por los que se produce este comportamiento es::

1. *Overhead*:
    a. Coste de creación de *threads* o procesos
    b. Coste de comunicación entre procesos o *threads*
    c. Coste de sincronización
2. Computación no paralelizable.
3. Procesadores inactivos.
4. Contención de recursos.

También, se puede observar que la reducción de tiempo de ejecución es más considerable en las series temporales de mayor tamaño, alcanzando el máximo valor en la serie temporal de mayor tamaño, Temperature en un 64%. El motivo es que cuanto mayor es la serie temporal, mayor número de cómputo deberá realizar el algoritmo secuencial. Este cómputo al realizar la paralelización, se reparte entre los distintos procesadores que intervienen en ella. Este se trata de un buen comportamiento del algoritmo paralelizado, ya que es una muestra de que la implementación llevada a cabo es escalable.

A continuación, se muestra un gráfico, donde se puede observar la evolución del *speed-up* real.

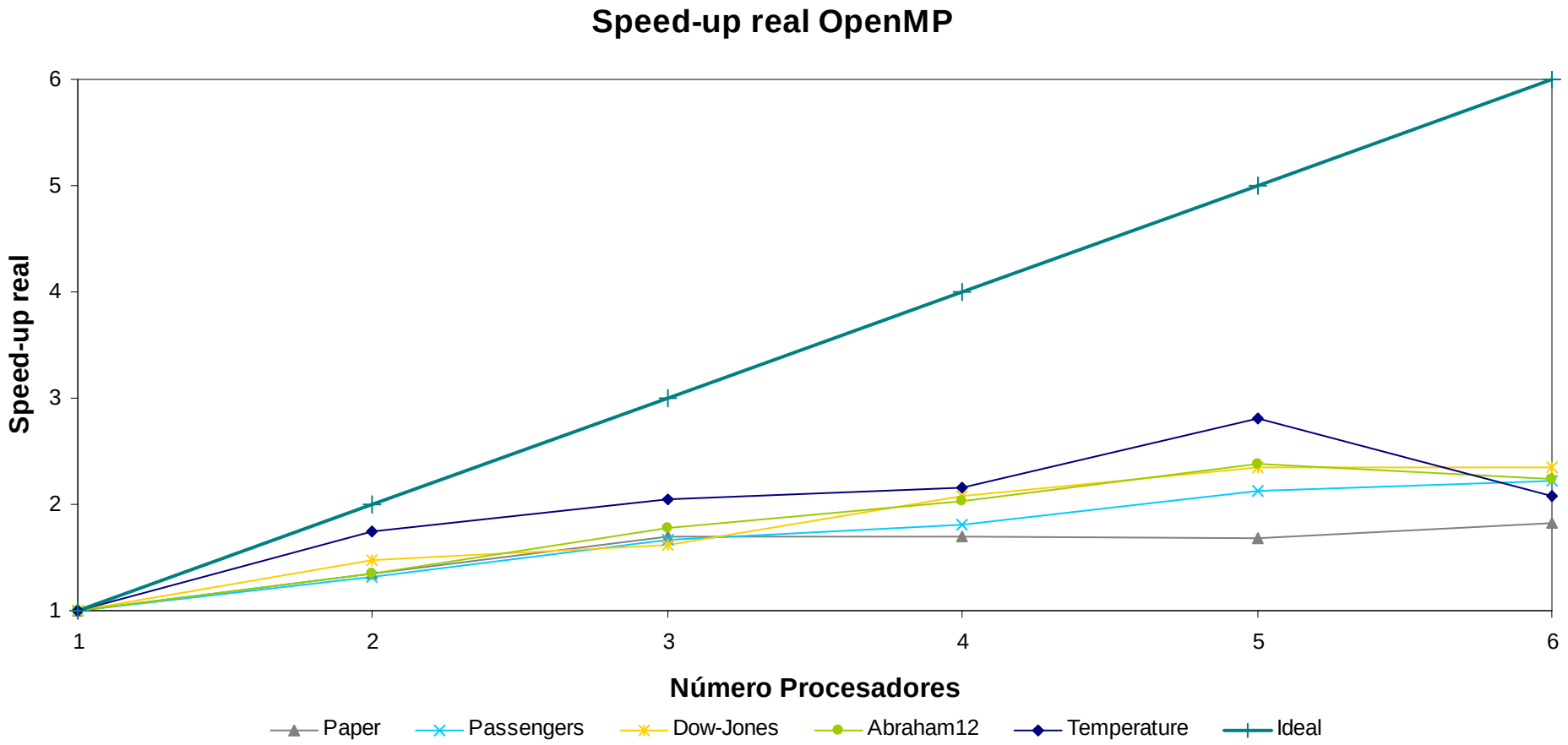

Figura 39 Comportamiento Speed-Up OpenMP

A continuación, se muestran los valores de la gráfica, para poder analizarlos:

#procesadores	Paper	Passengers	Dow-Jones	Abraham12	Temperature
1	1,00	1,00	1,00	1,00	1,00
2	1,36	1,32	1,47	1,35	1,75
3	1,70	1,66	1,61	1,77	2,05
4	1,69	1,80	2,09	2,03	2,16
5	1,68	2,12	2,35	**2,37**	**2,82**
6	**1,82**	**2,22**	**2,35**	2,24	2,08

Tabla 18 - Comportamiento *speed-up* OpenMP

Recordar que el *speed-up* se trata de la medida que cuantifica el beneficio de resolver el problema en paralelo. En función de lo que se considere como el tiempo de ejecución secuencial existen dos variaciones de esta medida.

1. ***speed-up* relativo**: el tiempo de ejecución secuencial usado es el tiempo total de ejecución del programa paralelo cuando éste es ejecutado sobre un único procesador del computador paralelo.
2. ***speed-up* real**: el tiempo de ejecución secuencial usado es el tiempo del mejor programa secuencial para resolver el problema.

$$S(n) = T(1)/T(n)$$

El *speed-up* representado en la gráfica al igual que en la experimentación de MPI es el *speed-up* real.

En la gráfica se puede observar el efecto comentado anteriormente, y es que para las series temporales de menor tamaño la mayor aceleración se alcanza con 6 procesadores, mientras que para las series temporales de mayor tamaño (Abraham12 y Temperature) se alcanza con cinco procesadores.

Pese a que el objetivo del proyecto es reducir al máximo el tiempo de ejecución, también es importante analizar el comportamiento de la paralelización implementada respecto a el aprovechamiento de los recursos disponibles, procesadores. Para ello, es interesante mostrar el comportamiento de la eficiencia en función de cómo se vaya incrementando el número de procesadores.

En muchas ocasiones, un algoritmo secuencial es evaluado en términos de su tiempo de ejecución, expresado como una función del tamaño de su entrada. El tiempo de ejecución de un sistema paralelo depende no sólo del tamaño de su entrada sino también de la arquitectura paralela, el número de procesadores y característica de la máquina tales como: velocidad del procesador, velocidad de los canales de comunicación, topología de interconexión y técnicas de ruteo. Por este motivo no puede evaluarse un algoritmo paralelo aisladamente de la arquitectura paralela sobre la que es implementado.

Un algoritmo que posee una buena performance para un problema seleccionado o sobre un número determinado de procesadores en una máquina dada puede funcionar pobremente si alguno de los parámetros cambia. La escalabilidad de un algoritmo paralelo sobre una arquitectura es una medida de su habilidad de obtener *speed-up* creciente linealmente con respecto al número de procesadores, en consecuencia refleja la capacidad del sistema paralelo para usar efectivamente una cantidad creciente de recursos de procesamiento.

En la siguiente gráfica, se puede observar cuál es el comportamiento de la eficiencia del algoritmo paralelizado para cada una de las series temporales empleadas en la experimentación:

#procesadores	Paper	Passengers	Dow-Jones	Abraham12	Temperature
1	100,00	100,00	100,00	100,00	100,00
2	67,68	73,74	66,04	87,51	67,85
3	59,12	53,72	55,31	68,39	56,73
4	50,68	52,15	45,11	54,03	42,37
5	47,46	46,99	42,46	56,31	33,64
6	37,27	39,17	37,08	34,73	30,34

Tabla 19 - Comportamiento eficiencia en OpenMP

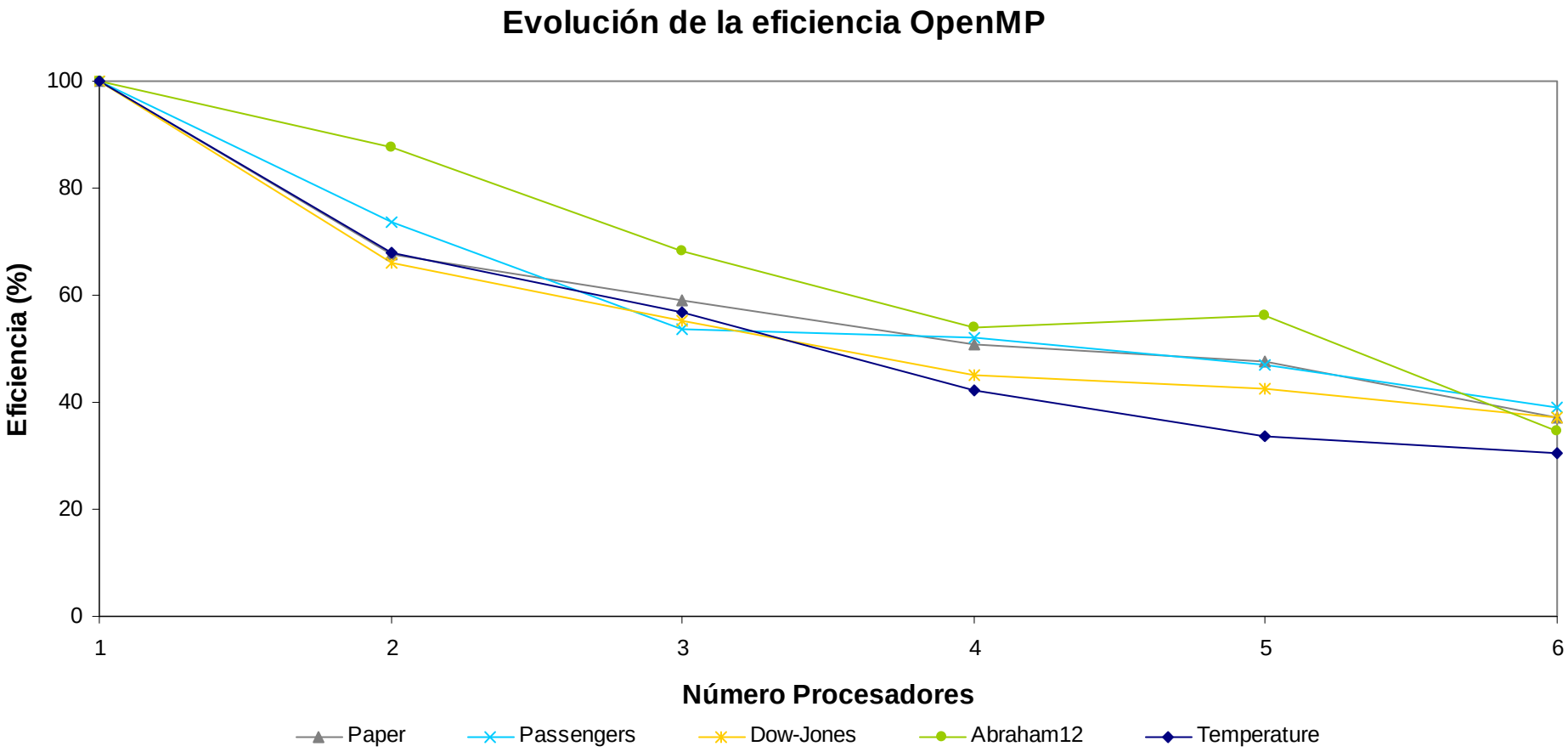

Figura 40 Comportamiento Eficiencia OpenMP

Recordemos que la eficiencia de un programa paralelo, como su nombre indica, expresa el aprovechamiento de los recursos de procesador, por parte de dicho programa. Se puede definir como:

$$E(n) = \frac{S(n)}{n} = \frac{T(1)}{nT(n)}$$

De donde se deduce que, para el caso óptimo de *speed-up* lineal la eficiencia toma el valor 1.

Se observa un comportamiento en la gráfica, y es que la eficiencia tiende a aumentar conforme aumenta el tamaño del problema. Lo cual se debe a que el problema secuencial, cuanto mayor tamaño tenga las series temporales, mayor número de operaciones deberá realizar, que al lograr la paralelización, se conseguirá que cierto número de operaciones ejecutadas en secuencial, se puedan ejecutar en paralelo en los distintos procesadores que intervienen.

También, se puede observar en la gráfica, que la eficiencia tiende a disminuir conforme se aumentan el número de procesos.

5.3 Comparativa MPI y OpenMP

En el siguiente apartado se realizará un análisis comparativo entre la implementación realizada con MPI y OpenMP.

Para poder realizar una comparativa entre MPI y OpenMP, es necesario mencionar, que a la hora de realizar la paralelización con ambos paradigmas, se ha realizado la misma estrategia de descomposición del problema, es decir, se han paralelizado los mismos fragmentos de código en ambos paradigmas. De esta forma, se puede garantizar que la comparativa a realizar puede dar resultados que ayuden a tomar una decisión sobre la implementación más adecuada para el problema.

Se comenzará realizando para cada una de las series temporales un análisis del comportamiento del tiempo de ejecución.

A continuación, se muestran los tiempos de ejecución para la serie temporal Paper:

#procesadores	Paper-MPI	Paper-OpenMP	Dif.(%) MPI/OpenMP
1	16.594,64	16.594,64	0%
2	13.442,47	12.228,18	10%
3	10.149,00	9.751,35	4%
4	9.734,47	9.792,42	-1%
5	8.183,67	9.866,26	-17%
6	10.376,34	9.116,08	14%

Tabla 20 - Comparativa tiempo ejecución MPI/OpenMP serie temporal Paper

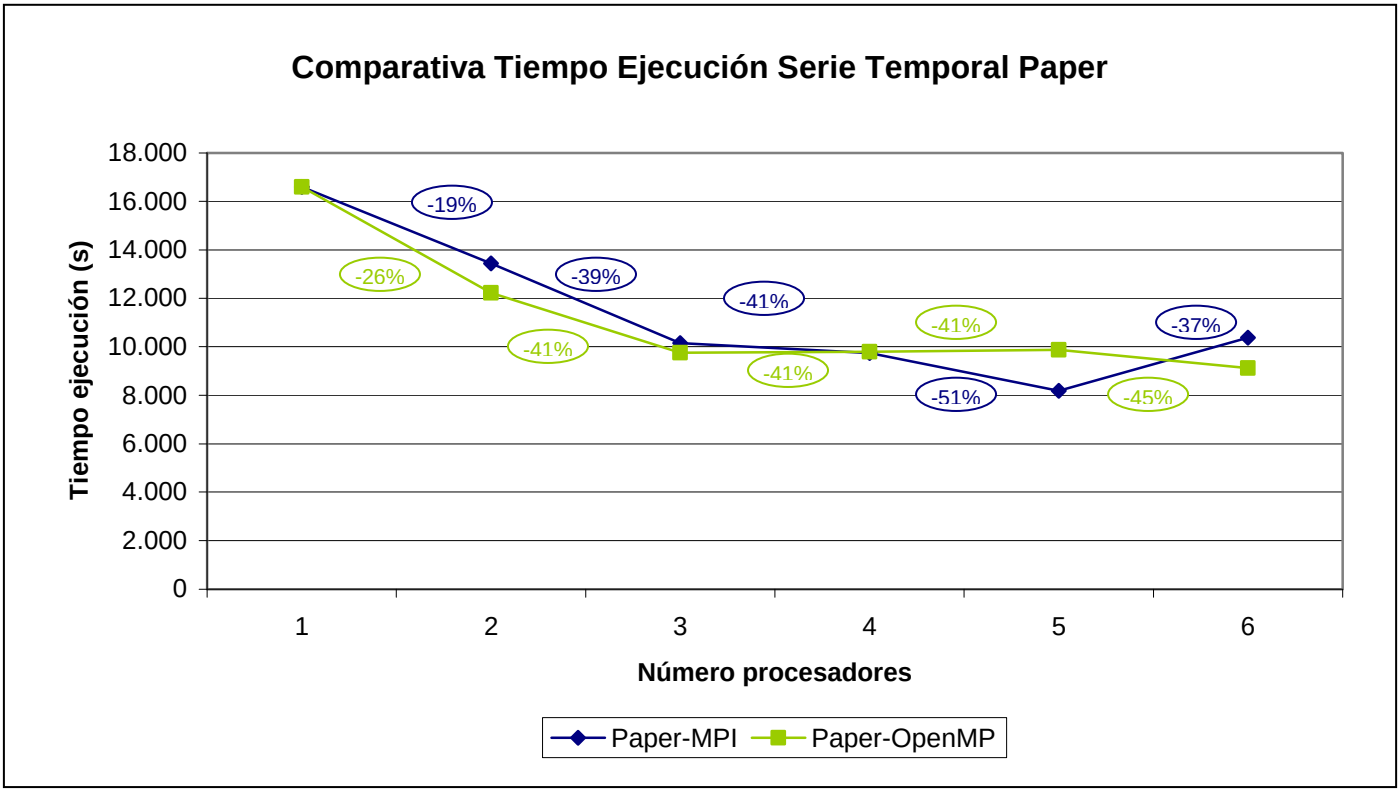

Figura 41 Comparativa MPI/OpenMP Tiempo Ejecución serie temporal Paper

En el caso de la serie temporal de menor tamaño, el caso de Paper que tiene un tamaño de 101 elementos, se observa que el tiempo de ejecución tiene mejor comportamiento con OpenMP para dos y tres procesadores. Sin embargo, el mejor resultado en tiempo de ejecución obtenido para la serie temporal Paper es con MPI empleando cinco procesadores 51%. Al añadir un sexto procesador, MPI sufre un rebote incrementando el tiempo de ejecución mientras que OpenMP con el incremento de procesadores sufre una caída más suave en el tiempo de ejecución, pero logra permanecer más constante.

A continuación, se muestran y representan los tiempos obtenidos para la serie temporal de Passengers.

#procesadores	Passengers - MPI	Passengers - OpenMP	Dif.(%) MPI/OpenMP
1	28.506,78	28.506,78	0%
2	19.211,39	21.581,92	-11%
3	16.826,84	17.181,38	-2%
4	13.559,48	15.796,87	-14%
5	12.177,58	13.426,04	-9%
6	11.054,15	12.814,13	-14%

Tabla 21 - Comparativa tiempo ejecución MPI/OpenMP serie temporal Passengers

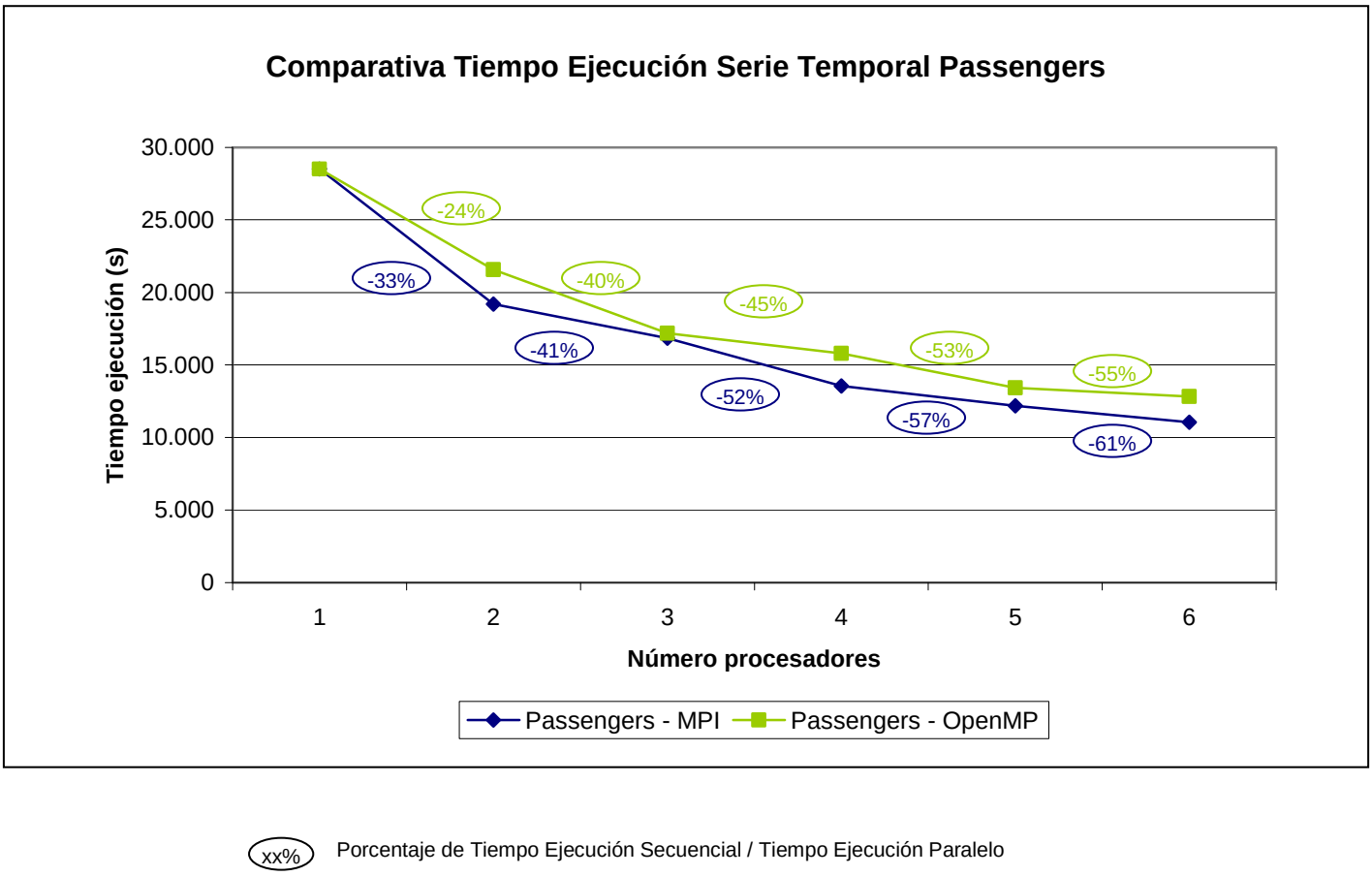

Figura 42 Comparativa Tiempo Ejecución MPI/OpenMP serie temporal Passengers

En la serie temporal Passengers se vuelve a repetir un comportamiento, y es que OpenMP, sufre una reducción en tiempo de ejecución más suave que MPI. Sin embargo, en este caso, MPI muestra para todos los procesadores mejor comportamiento, alcanzando el mínimo con seis procesadores llegando a obtener una mejora de un 61%.

A continuación, se muestran y representan los tiempos obtenidos para la serie temporal de Dow-Jones.

#procesadores	Dow-Jones - MPI	Dow-Jones - OpenMP	Dif.(%) MPI/OpenMP
1	26.001,83	26.001,83	0%
2	21.015,81	17.630,27	19%
3	11.301,24	16.134,98	-30%
4	11.977,87	12.465,64	-4%
5	10.138,44	11.068,11	-8%
6	10.756,16	11.062,90	-3%

Tabla 22 - Comparativa tiempo ejecución MPI/OpenMP serie temporal Dow-Jones

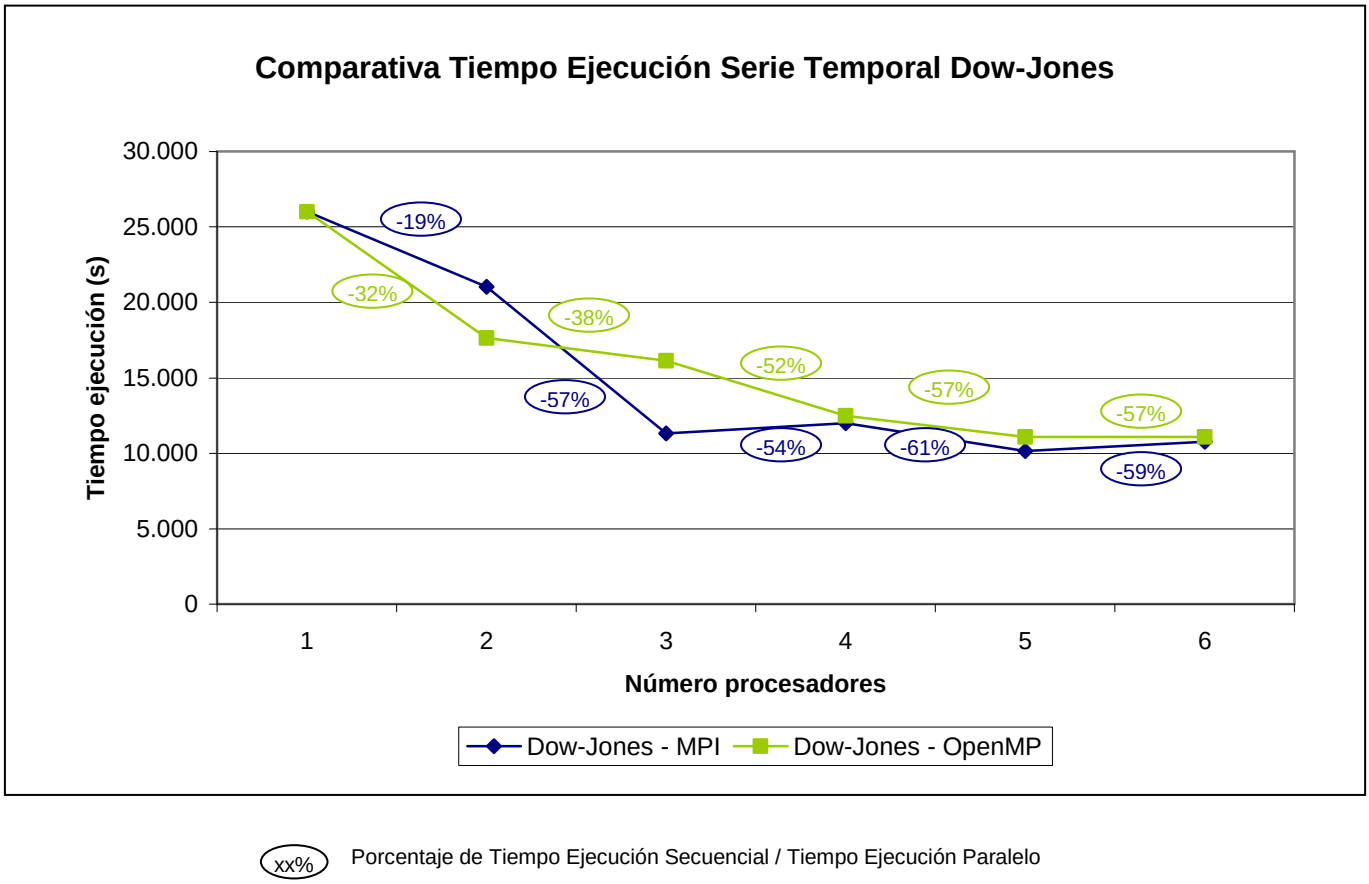

Figura 43 Comparativa Tiempo Ejecución MPI/OpenMP serie temporal Dow-Jones

En la serie temporal de Dow-Jones, nuevamente se obtiene como mejor resultado en el tiempo de ejecución con MPI empleando cinco procesadores alcanzando una mejora del 61%. En la gráfica, se puede observar que OpenMP, tiene una caída importante realizando la paralelización con dos procesadores, pero a continuación su reducción es mucho más suave que para MPI.

A continuación, se muestran y representan los tiempos obtenidos para la serie temporal de Abraham12.

#procesadores	Abraham12 - MPI	Abraham12 - OpenMP	Dif.(%) MPI/OpenMP
1	34.297,52	34.297,52	0%
2	24.977,65	25.336,81	-1%
3	22.656,25	19.337,18	17%
4	18.560,53	16.919,31	10%
5	13.806,02	14.452,57	-4%
6	14.440,66	15.335,49	-6%

Tabla 23 - Comparativa tiempo ejecución MPI/OpenMP serie temporal Abraham12

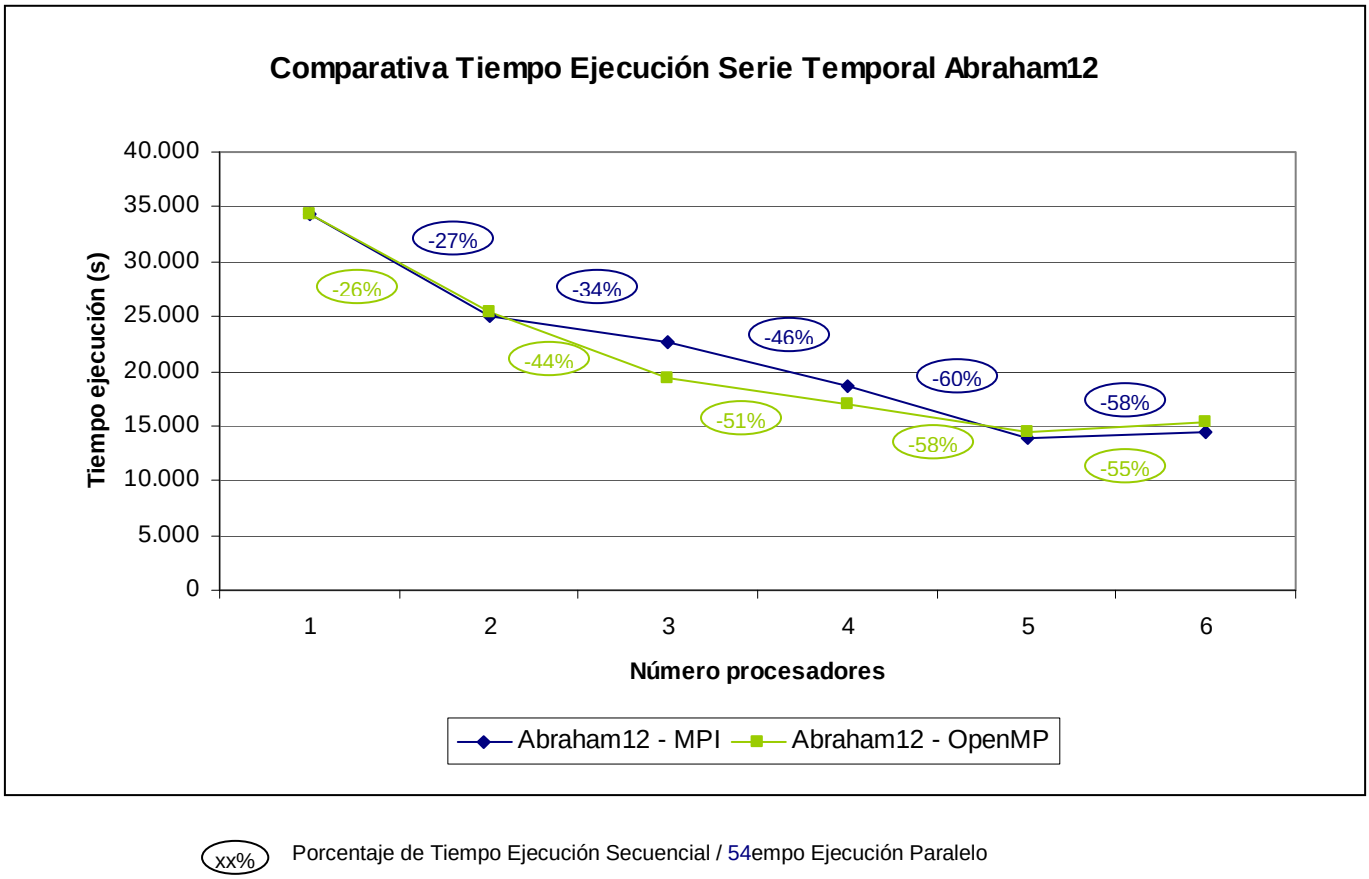

Figura 44 Comparativa Tiempo Ejecución MPI/OpenMP serie temporal Abraham12

Como se puede observar en la gráfica, y viene siendo común en el resto de series temporales, OpenMP inicialmente tiene un mejor comportamiento en el tiempo de ejecución, pero se vuelve a obtener el mejor resultado con MPI empleando cinco procesadores alcanzando un 60% de mejora. En un inicio la caída de OpenMP hasta los tres procesadores es más brusca que en MPI pero a partir de estos tres procesadores, la caída es más suave que para MPI.

#procesadores	Temperature - MPI	Temperature - OpenMP	Dif.(%) MPI/OpenMP
1	55.582,04	55.582,04	0%
2	46.143,62	31.759,01	45%
3	25.478,07	27.090,26	-6%
4	21.884,49	25.717,15	-15%
5	15.748,41	19.742,84	-20%
6	22.049,88	26.672,60	-17%

Tabla 24 - Comparativa tiempo ejecución MPI/OpenMP serie temporal Temperature

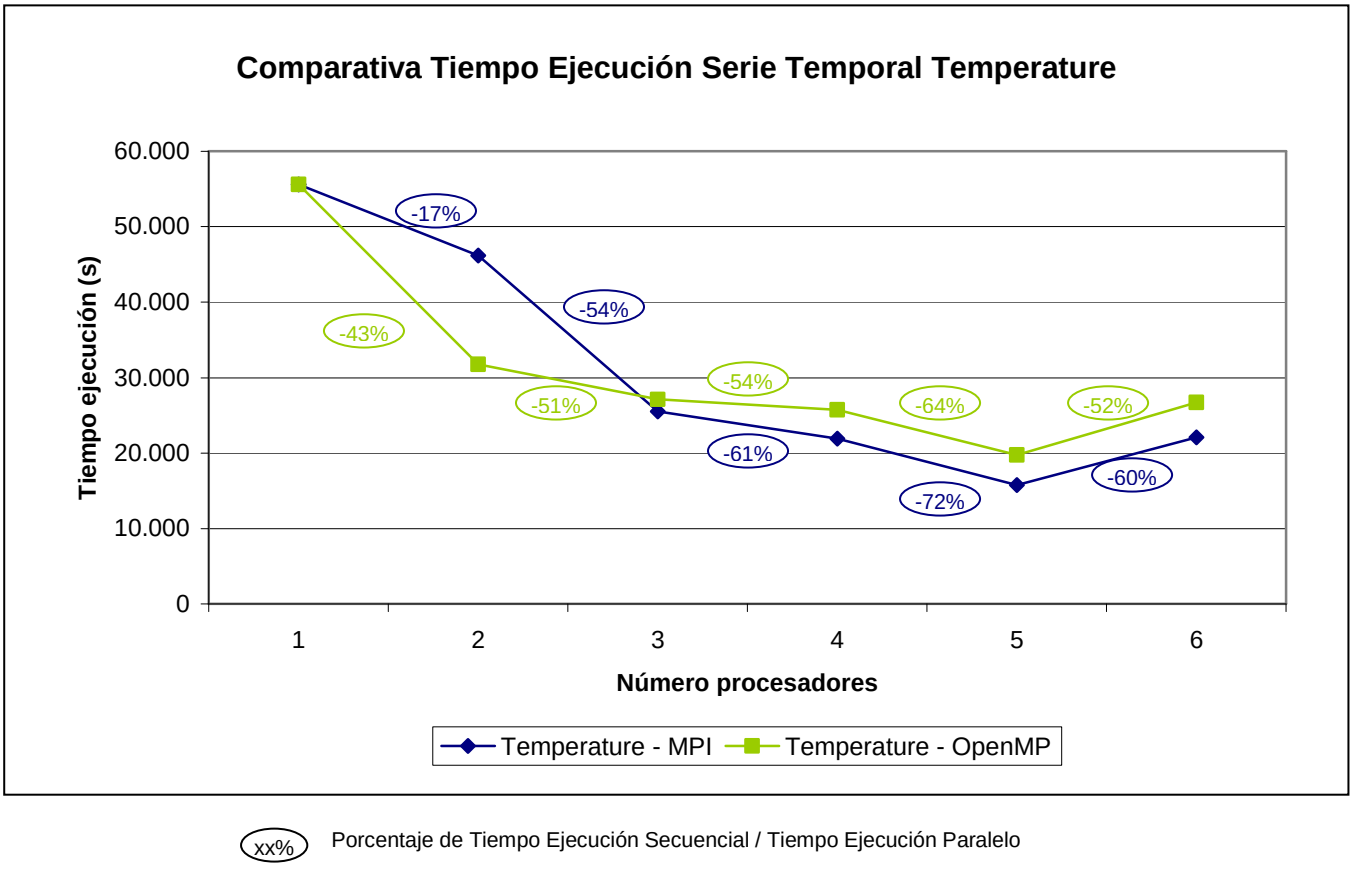

Figura 45 Comparativa tiempo ejecución MPI/OpenMP serie temporal Temperature

En la serie temporal de mayor tamaño, nuevamente se obtiene el mejor resultado de tiempo con el paradigma MPI empleando cinco procesadores, en donde se llega a obtener un 72% de mejora en cuanto a tiempo. Nuevamente, se obtienen mejores tiempos hasta tres procesadores con OpenMP, a partir del cual, la caída en el tiempo de ejecución en OpenMP es más suave que en MPI.

Se puede observar, que la mejora de tiempo obtenida en la serie temporal de Temperature es considerablemente mayor al resto, puesto que como ya se

comentó, cuanto mayor es la serie temporal, la mejora de la paralelización se hace más apreciable.

Al igual que se realizado la comparativa entre la paralelización con MPI y OpenMP en cuanto a tiempo de ejecución, se va a realizar la comparativa considerando la eficiencia.

Ya que la eficiencia representa el porcentaje de tiempo empleado en proceso efectivo, nos ayudará a analizar no sólo que paradigma de programación presenta mejor comportamiento en cuanto a tiempo de ejecución, sino que además, la eficiencia nos dará información del paradigma que aprovecha en mayor medida los recursos disponibles en la experimentación.

A continuación, se muestran los datos de eficiencia tanto para MPI como para OpenMP:

#procesadores	Paper - MPI	Paper - OpenMP
1	100,00	100,00
2	61,72	67,85
3	54,50	56,73
4	42,62	42,37
5	40,56	33,64
6	26,65	30,34

Tabla 25 - Comparativa eficiencia MPI/OpenMP serie temporal Paper

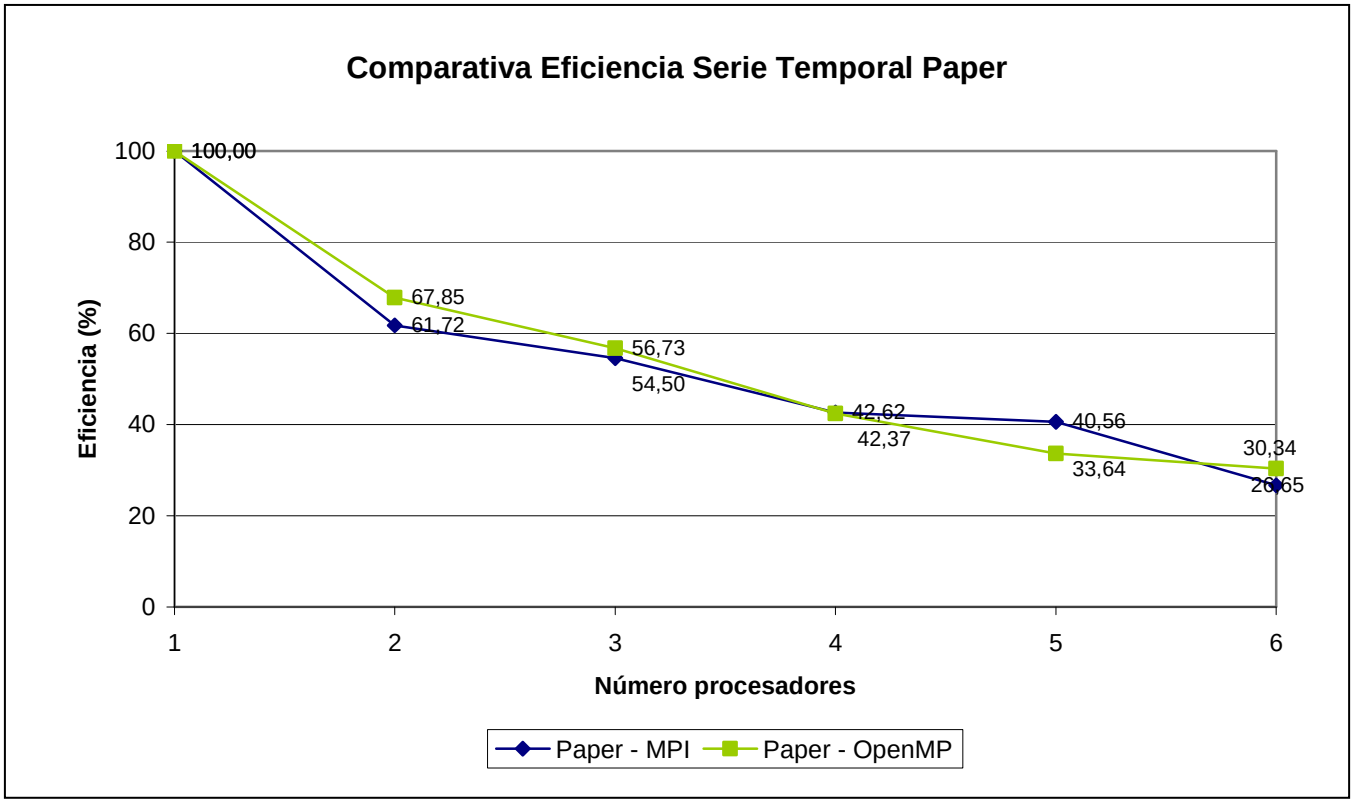

Figura 46 Comparativa eficiencia MPI/OpenMP serie temporal Paper

En la serie temporal Paper se puede observar que no existen grandes diferencias en cuanto a eficiencia, presenta mejores cifras OpenMP que MPI salvo para cinco procesadores en donde MPI da un pequeño salto de eficiencia. Como se puede observar en la gráfica, la eficiencia va descendiendo según se van incrementando el número de procesadores. Este descenso se debe a que existen intervalos de tiempo en los que los procesadores permanecen inactivos. La causa de esta inactividad, puede deberse a distintas circunstancias tales como: tiempos de sincronización, tiempos de comunicación, tiempos debido a la contención de recursos.

Se continúa analizando los resultados obtenidos en eficiencia en la serie temporal de Passengers:

#procesadores	Passengers - MPI	Passengers - OpenMP
1	100,00	100,00
2	74,19	66,04
3	56,47	55,31
4	52,56	45,11
5	46,82	42,46
6	42,98	37,08

Tabla 26 - Comparativa eficiencia MPI/OpenMP serie temporal Passengers

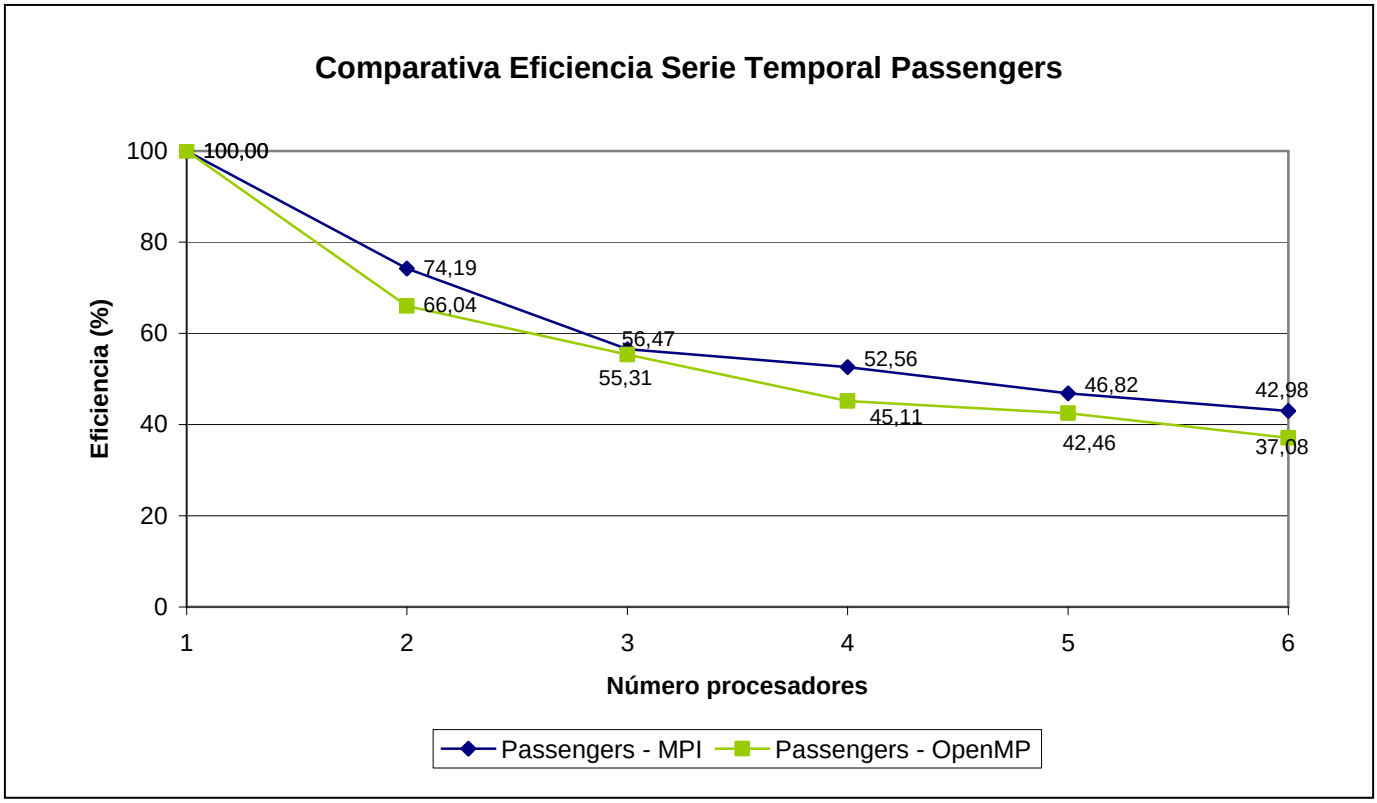

Figura 47 Comparativa eficiencia MPI/OpenMP serie temporal Passengers

Para la serie temporal Passengers, como se puede observar en la gráfica MPI obtiene mejores resultados. Es importante destacar que la eficiencia se puede comprobar que reduce tan bruscamente como en el caso de la serie temporal Paper.

Se continúa analizando los resultados obtenidos en eficiencia en la serie temporal de Dow-Jones:

#procesadores	Dow-Jones - MPI	Dow-Jones - OpenMP
1	100,00	100,00
2	61,86	73,74
3	76,69	53,72
4	54,27	52,15
5	51,29	46,99
6	40,29	39,17

Tabla 27 - Comparativa eficiencia MPI/OpenMP serie temporal Dow-Jones

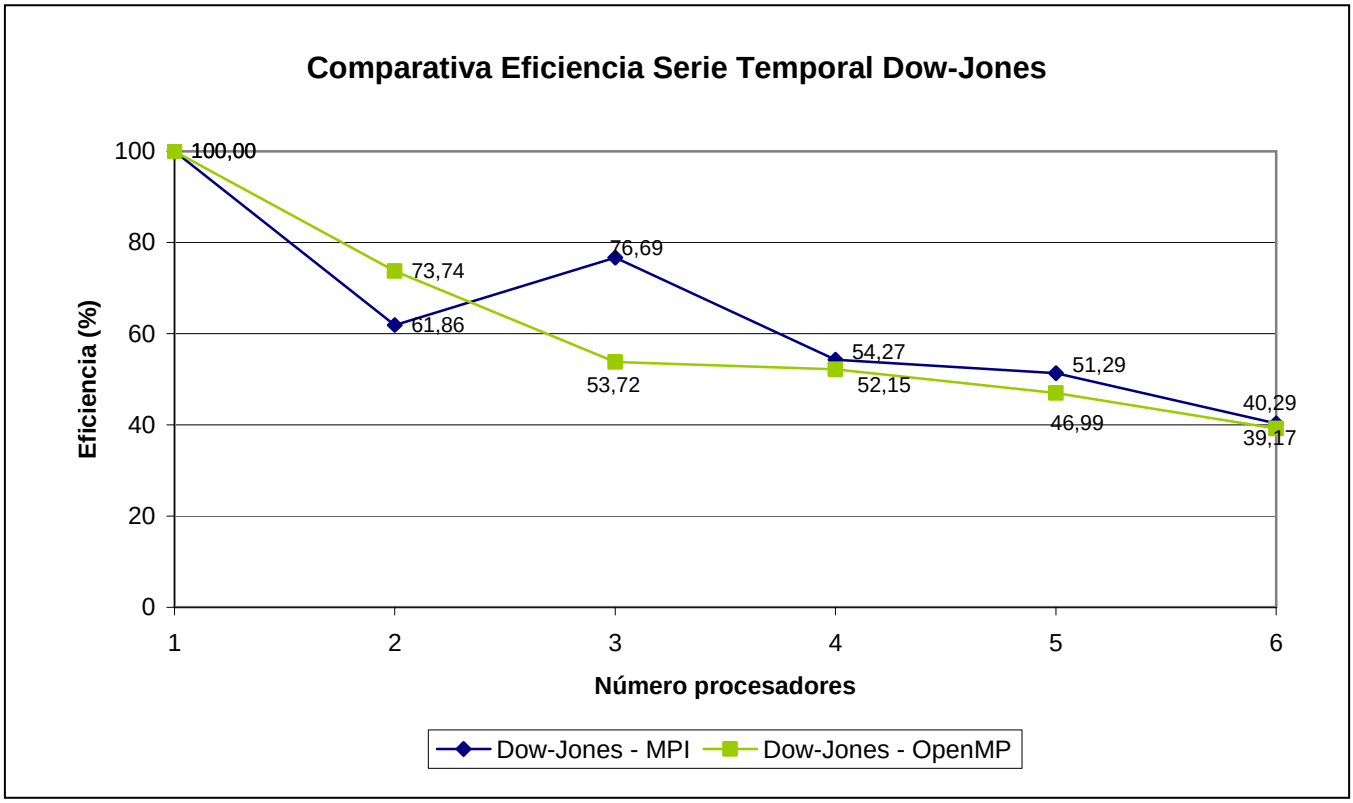

Figura 48 Comparativa eficiencia MPI/OpenMP serie temporal Dow-Jones

Como se puede observar en la gráfica, a partir de dos procesadores MPI presenta una mayor eficiencia respecto a OpenMP. Destacar el buen resultado en cuanto a eficiencia obtenido por MPI con tres procesadores en donde llega a alcanzar un 70% mientras que OpenMP no alcanza el 54%.

Se continúa analizando los resultados obtenidos en eficiencia en la serie temporal de Abraham12:

#procesadores	Abraham12 - MPI	Abraham12 - OpenMP
1	100,00	100,00
2	68,66	67,68
3	50,46	59,12
4	46,20	50,68
5	49,68	47,46
6	39,58	37,27

Tabla 28 - Comparativa eficiencia MPI/OpenMP serie temporal Abraham12

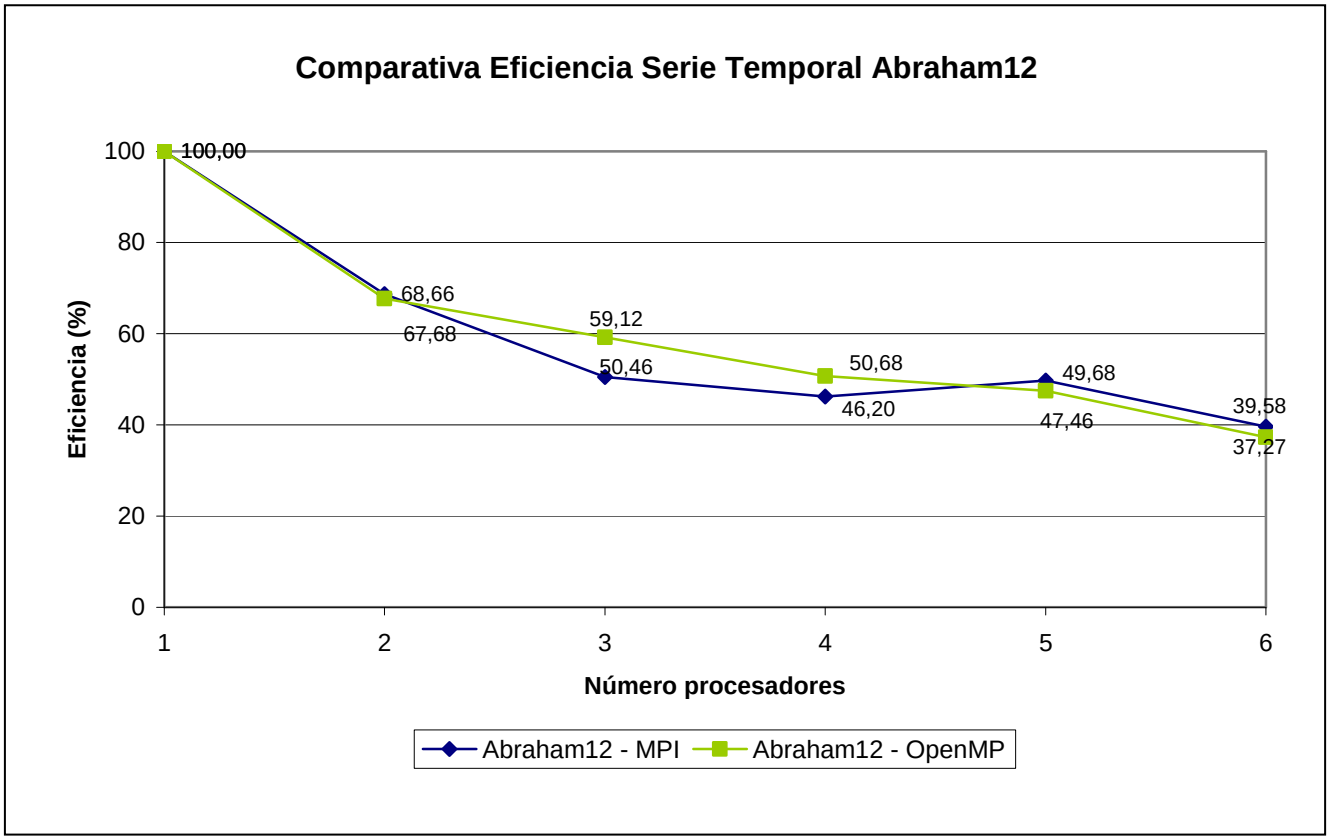

Figura 49 Comparativa eficiencia MPI/OpenMP serie temporal Abraham12

Se puede observar en la gráfica que la diferencia de eficiencia para la serie temporal Abraham12 entre OpenMP y MPI es muy reducida. Presentando su mayor diferencia en tres procesadores, donde OpenMP presenta una mejora de un 9% respecto a MPI. Pero según se van ampliando el número de procesadores, esta diferencia se va reduciendo llegando a mejorar MPI la eficencia de OpenMP haciendo uso de seis procesadores.

Se continúa analizando los resultados obtenidos en eficiencia en la serie temporal de mayor tamaño, Temperature:

#procesadores	Temperature - MPI	Temperature - OpenMP
1	100,00	100,00
2	60,23	87,51
3	72,72	68,39
4	63,49	54,03
5	70,59	56,31
6	42,01	34,73

Tabla 29 - Comparativa eficiencia MPI/OpenMP serie temporal Temperature

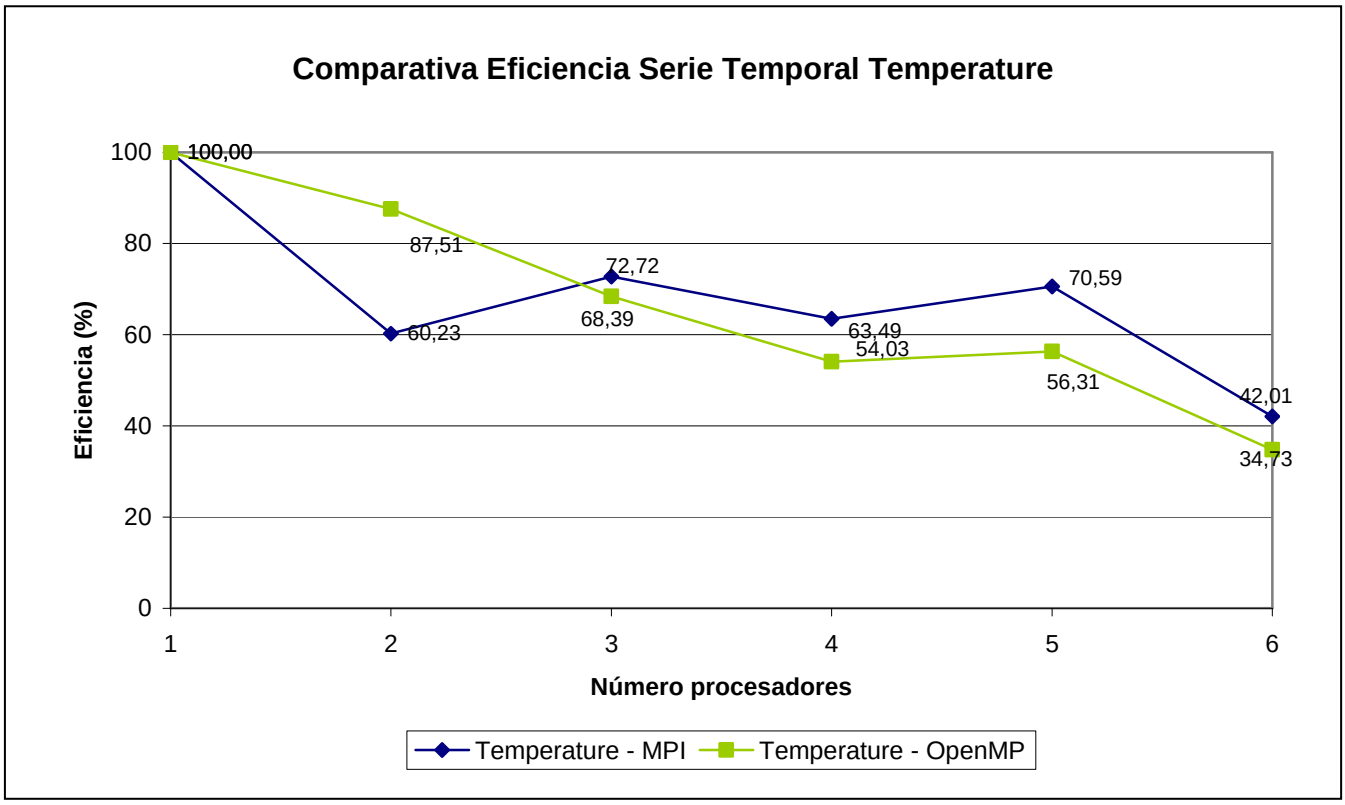

Figura 50 Comparativa eficiencia MPI/OpenMP serie temporal Temperature

La serie temporal Temperature que se trata de la de mayor tamaño, presenta unos resultados de eficiencia bastante buenos. Recordemos que la eficiencia decae al incrementar el número de procesadores. Por el contrario, para una instancia más grande del mismo problema se tiene un mejor *speed-up* (y su eficiencia) para el mismo número de procesadores.

Comparando el comportamiento de la eficiencia para la serie temporal de mayor tamaño como es el caso de Temperature con el resto de serie temporales, se puede comprobar que incrementar el número de procesadores reduce la eficiencia y crecer el tamaño del problema aumenta.

6. Conclusiones Generales

En el presenta capítulo, después de haberse realizado el análisis de los resultados obtenidos con la paralelización y el contraste de éstos entre MPI y OpenMP, se mostrarán las conclusiones finales.

Estas conclusiones dependerán de los criterios que se quieran aplicar sobre el problema, y se tendrán en cuenta las distintas variables que influyen en los resultados del problema.

En primer lugar, se van a mostrar los resultados obtenidos en cuanto a tiempo de ejecución para cada uno de los paradigmas. Considerando únicamente el tiempo de ejecución las conclusiones que se pueden sacar son:

- Dado el comportamiento del tiempo de ejecución para el paradigma MPI, se muestran las Δtiempo respecto al tiempo de ejecución secuencial, de acuerdo a la siguiente expresión:

$$\Delta tiempo = \frac{\left(T_{paralelo} - T_{\sec uencial}\right)}{T_{\sec uencial}} \times 100$$

ΔTiempo Paper	ΔTiempo Passengers	ΔTiempo Dow-Jones	ΔTiempo Abraham12	ΔTiempo Temperature	Promedio (ΔTiempo)
0%	0%	0%	0%	0%	0%
-19%	-33%	-19%	-27%	-17%	-23%
-39%	-41%	-57%	-34%	-54%	-45%
-41%	-52%	-54%	-46%	-61%	-51%
-51%	-57%	-61%	-60%	-72%	-60%
-37%	-61%	-59%	-58%	-60%	-55%

Tabla 30 - Datos de mejora de tiempo de ejecución en MPI

Como se puede observar en la tabla, los mejores resultados en cuanto a tiempo de ejecución con MPI es empleando cinco procesadores, en donde se alcanza de promedio un 60% de mejora respecto al tiempo de ejecución secuencial. Como se ha comentado en el análisis realizado de los resultados, se puede observar que para cinco procesadores, las series temporales de mayor tamaño muestran una mayor mejora en tiempo de ejecución que las series temporales de menor tamaño.

- Dado el comportamiento del tiempo de ejecución para el paradigma OpenMP, se muestran las Δtiempo respecto al tiempo de ejecución secuencial, de acuerdo a la siguiente expresión:

$$\Delta tiempo = \frac{\left(T_{paralelo} - T_{\sec uencial}\right)}{T_{\sec uencial}} \times 100$$

ΔTiempo Paper	ΔTiempo Passengers	ΔTiempo Dow-Jones	ΔTiempo Abraham12	ΔTiempo Temperature	Promedio (ΔTiempo)
0%	0%	0%	0%	0%	0%
-26%	-24%	-32%	-26%	-43%	-30%
-41%	-40%	-38%	-44%	-51%	-43%
-41%	-45%	-52%	-51%	-54%	-48%
-41%	-53%	-57%	-58%	-64%	-55%
-45%	-55%	-57%	-55%	-52%	-53%

Tabla 31 - Datos de mejora de tiempo de ejecución en OpenMP

Como se observa en la tabla, para el paradigma OpenMP, el número de procesadores que también muestra mejor comportamiento en el problema respecto al tiempo de ejecución son los cinco procesadores alcanzando un promedio de mejora de un 55%. Se puede observar, al igual que antes, que las series temporales de mayor tamaño muestran un mejor comportamiento que las series temporales de menor tamaño.

Observando las crifras para ambos paradigmas, se puede concluir diciendo que MPI presenta el mejor resultado en tiempo de ejecución, ya que logra alcanzar con cinco procesadores un promedio de mejora de un 60% frente al 55% de OpenMP empleando ese mismo número de procesadores.

Pero no sólo se ha analizado los resultados de la paralelización considerando el tiempo de ejecución, ya que, el tiempo de ejecución de un algoritmo paralelo depende no sólo del tamaño de su entrada sino también de la arquitectura paralela, el número de procesadores, y características de las máquinas. Por lo que, además del comportamiento respecto al tiempo de ejecución, es necesario analizar el comportamiento del algoritmo en cuanto a la eficiencia ya que nos muestra cómo aprovecha el algoritmo paralelo los recursos.

A continuación, se muestran los datos obtenidos en la experimentación considerando la eficiencia expresada en porcentaje para el paradigma MPI:

#procesadores	Paper	Passengers	Dow-Jones	Abraham12	Temperature	Promedio (Eficiencia)
1	100,00	100,00	100,00	100,00	100,00	100,00
2	61,72	74,19	61,86	68,66	60,23	65,33
3	54,50	56,47	76,69	50,46	72,72	62,17
4	42,62	52,56	54,27	46,20	63,49	51,83
5	40,56	46,82	51,29	49,68	70,59	51,79
6	26,65	42,98	40,29	39,58	42,01	38,30

Tabla 32 - Datos de eficiencia en MPI

Como se puede observar, la eficiencia se va reduciendo según se van incrementando el número de procesadores. Se puede observar que la mejora

de tiempo en ejecución obtenida con seis procesadores tiene un coste alto, ya que existe una eficiencia de apenas un 38%.

A continuación, se muestran los datos obtenidos de la experimentación considerando la eficiencia expresada en porcentaje para el paradigma OpenMP:

#procesa-dores	Paper	Passengers	Dow-Jones	Abraham12	Temperature	Promedio (Eficiencia)
1	100,00	100,00	100,00	100,00	100,00	100,00
2	67,85	66,04	73,74	67,68	87,51	72,57
3	56,73	55,31	53,72	59,12	68,39	58,65
4	42,37	45,11	52,15	50,68	54,03	48,87
5	33,64	42,46	46,99	47,46	56,31	45,37
6	30,34	37,08	39,17	37,27	34,73	35,72

Tabla 33 - Datos de eficiencia en OpenMP

Se puede observar también, que la eficiencia se reduce según se van incrementando el número de procesadores.

A la hora de tomar la decisión sobre la arquitectura más adecuada para el problema, se debe considerar que variable tiene mayor peso en la decisión si la mejora significativa en el tiempo de ejecución, visión que podría tener cualquier usuario del problema o un equilibrio entre tiempo de ejecución y eficiencia, que se trata de la visión que tendría un administrador de sistemas.

Considerando el equilibrio entre mejora en tiempo de ejecución y eficiencia considerando los datos obtenidos la arquitectura que parecería más adecuada es el uso del paradigma MPI considerando el uso de cinco procesadores ya que se obtiene una mejora de tiempo de un 60% y un eficiencia del 52%, lo cual significa un 56% como promedio de ambas variables.

Es interesante comentar que dependiendo del problema, es posible que existan otras configuraciones que presenten mejor comportamiento que la que se ha obtenido en la experimentación expuesta. Por la naturaleza del problema, en el que el mayor consumo que realiza el Sistema es de CPU y prácticamente no requiere hacer uso de memoria, MPI muestra buenos resultados. Sin embargo, en el caso en el que el problema a abordar requiriese un mayor consumo de memoria que de CPU, sería muy probable que la mejora obtenida gracias al uso de OpenMP y del uso de la memoria compartida sería considerable.

Como se ha comentado en el apartado de experimentación, las pruebas se han lanzado dentro de una arquitectura multiprocesador. Más adelante, dentro del apartado de Trabajos Futuros, se comenta que sería interesante poder realizar la experimentación llevada a cabo dentro de una arquitectura multicomputador. Sería interesante observar los tiempos obtenidos en dicha arquitectura, puesto que por la naturaleza del problema, debido a que requiere realizar escritura en disco de diversos ficheros de configuración para su posterior uso en SNNS, dicha escritura en disco probablemente suponga un cuello de botella que podrá

ser salvado en parte gracias al uso de una arquitectura multicomputador, puesto que la contención de recursos se reduciría.

Después de haber realizado la implementación de la paralelización con MPI (paso de mensajes) y OpenMP (memoria compartida), se ha considerado interesante realizar una reflexión acerca de las ventajas/desventajas de éstos.

En primer lugar, comentar que OpenMP inicialmente presenta muchas mayores facilidades a la hora de realizar la implementación puesto que el compilador realiza muchas funciones que con MPI es necesario realizar implícitamente en código como puede ser el reparto de carga de trabajo o muchas barreras de sincronización que OpenMP lleva implícita en muchas directivas, mientras que en MPI se necesita implementar dentro del código.

OpenMP, ofrece mayores facilidades a la hora de implementar, ya que además permite que la misma versión del programa se pueda ejecutar en versión secuencial como en versión paralela ya que las directivas empleadas por OpenMP, se consideran como comentarios en el caso de lanzar la ejecución secuencial.

El hecho de que presente mayores facilidades a la hora de implementar, permite que se trate de un código generalmente más fácil de comprender, y por lo tanto, más fácil de mantener.

Como ventaja de OpenMP, es que MPI tiene su performance limitado por las redes de comunicación entre los distintos nodos que participen en la paralelización mientras que OpenMP al hacer uso de memoria compartida no se ve afectado por este hecho. Sin embargo, una gran desventaja de OpenMP frente a MPI es que OpenMP únicamente sólo se puede emplear en arquitecturas con memoria compartida, no es posible hacer uso de OpenMP en arquitectura multicomputadores. Sin embargo, MPI permite emplearse tanto en arquitectura compartidas como en arquitecturas multicomputadores.

OpenMP además, presenta la limitación que únicamente se pueden paralelizar lazos. Lo cual, puede ser en muchos casos un punto débil a la hora de realizar la paralelización de ciertos algoritmos. Además, puesto que OpenMP transparenta como se ha comentado anteriormente ciertos aspectos, no permite definir al más bajo nivel la paralización en ciertos aspectos. Lo cual concluye en que MPI pueda aplicarse a una gama más amplia de problemas.

Otro punto a considerar, es que OpenMP implementa la paralización mediante *threads*, mientras que MPI realiza la paralización haciendo uso de procesos. Puesto que el tiempo de creación de un *thread* en un proceso existente es menor que el tiempo de creación de un proceso nuevo, OpenMP presentará menor *overhead* en este aspecto. Al igual, que el tiempo requerido para terminar un thread que es menor al tiempo requerido para terminar un proceso. Sin embargo, el uso de *threads* presenta el inconveniente que al poder emplear memoria compartida dentro del mismo proceso, es necesario realizar una importante tarea de sincronización de *threads* para evitar la inconsistencia de

datos. Mientras que en el caso de MPI cada proceso dispone de sus variables privadas, y esta tarea de sincronización es menor.

Una ventaja adicional de MPI es que generalmente es mucho menos costoso, montar redes con mayor número de computadores, que incrementar la memoria de un sistema. Además, OpenMP al ejecutarse en un sistema multiprocesador, provoca que OpenMP en muchos problemas sea mucho más sensible a la contención de recursos que MPI.

7. Trabajos Futuros

1. Pruebas algoritmo paralelizado MPI en sistema multicomputador

Como se ha comentado anteriormente, el entorno de pruebas en el que se ha realizado la experimentación para MPI como para OpenMP se trata de un sistema multiprocesador. Sin embargo, el posible cuello de botella existente debido a la escritura en disco de cada uno de los procesos o *threads* podría ser reducido haciendo uso de un sistema multicomputador.

2. MPI + OpenMP

Realizando uso tanto de MPI como OpenMP conjuntamente se podría aprovechar las ventajas de la capacidad de realizar paralelización de multicomputadores que ofrece MPI junto con las ventajas de la paralelización mediante memoria compartida de OpenMP.

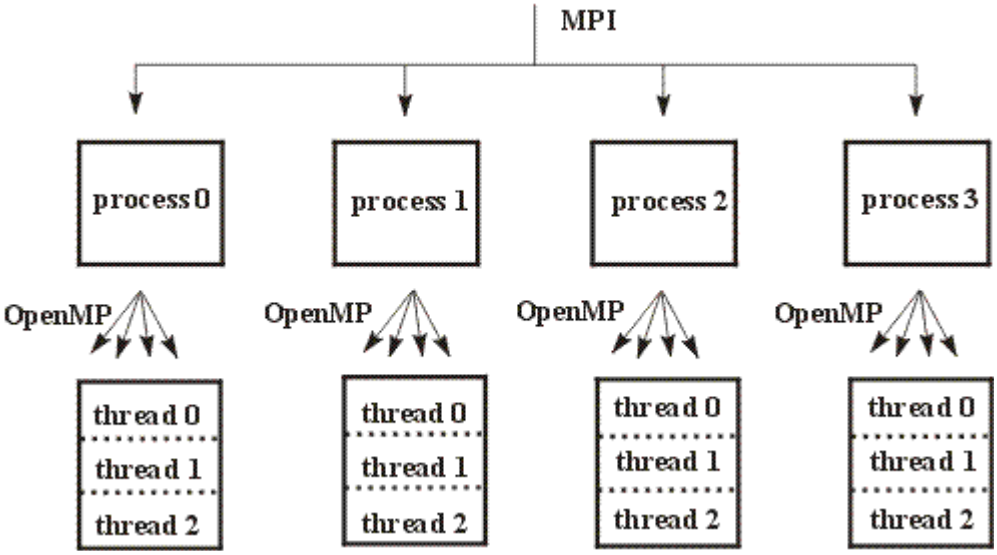

Figura 51 Híbrido MPI+OpenMP

3. Nuevas estrategias paralelización AGP

La estrategia de implementación empleada a la hora de paralelizar los algoritmos ha sido el mismo, el algoritmo maestro-esclavo. La decisión de haber empleado esta arquitectura se ha debido a que se desea obtener los mismos resultados en el Sistema que la versión secuencial de éste.

Sin embargo, existen otras estrategias de implementar una versión paralela del Sistema que se pasan a detallar:

- **Algoritmo de grano fino**: este tipo de algoritmo ha sido diseñado para ser implementado en multicomputadores masivamente paralelos. La población

se encuentra dividida espacialmente entre los distintos procesadores e, idealmente, cada procesador debería albergar un único individuo. El cruce y la selección de individuos se hará entre individuos que pertenezcan al mismo vecindario, formado por un conjunto de individuos adyacentes según la representación espacial antes citada.

- **Algoritmo de grano grueso**: las características más importantes de estos algoritmos son el uso de múltiples poblaciones y la migración de individuos entre ellas. Dado que una de las poblaciones evoluciona independientemente, el ratio de migración será muy importante de cara a obtener resultados satisfactorios.

- **Algoritmos híbridos**: quizás sea la estrategia de paralelización más compleja dado que combinan dos de los algoritmos anteriores en la estructura de dos niveles. En el nivel superior suele haber siempre un algoritmo de grano grueso (multipoblación). En el nivel inferior se han probado los tres tipos de algoritmos vistos: maestro-esclavo, grano fino y grano grueso. A esta estructuración en dos niveles se le denomina jerarquía y, por eso, muchas veces a estos algoritmos se les denomina algoritmos genéticos jerárquicos.

La primera posibilidad es combinar un algoritmo multipoblación en el nivel superior con un algoritmo de grano fino en el nivel inferior. Existen varias implementaciones de este tipo de algoritmos.

Como segunda alternativa para implementar algoritmos híbridos se puede usar un algoritmo multipoblacional en el nivel superior y un algoritmo maestro-esclavo en cada una de las poblaciones.

La tercera opción de hibridación es usar algoritmos multipoblacional en ambos niveles (superior e inferior). Usando una topología altamente conectada en el nivel inferior y una frecuencia de migraciones elevada se consigue una mejor distribución de las soluciones óptimas a través de las poblaciones. En el nivel superior se suele optar por frecuencias de migraciones bajas, lo cual hace que la complejidad no crezca demasiado, siendo aproximadamente igual a la de un algoritmo paralelo con multipoblaciones tradicional.

4. Nuevas posibilidades de *schedule* en OpenMP

En la experimentación realizada empleando OpenMP siempre se ha considerado un *schedule dynamic* y *chunk* igual a uno. Sin embargo, OpenMP tal y como se ha comentado anteriormente, permite emplear políticas de reparto de carga y *schedule* diferentes. Por lo que, podría ser interesante realizar la experimentación de OpenMP considerando otras opciones diferentes, ya que dependiendo del dominio del problema el parámetro *chunk* que define el tamaño de los bloques de carga que se reparte entre los distintos procesadores, puede ser una variable que tenga influencia en los tiempos de ejecución, y por tanto, también en eficiencia del algoritmo.

8. Gestión del Proyecto

En el presente capítulo se van a detallar los aspectos relacionados con la gestión del proyecto. Se mostrará la planificación (cómo ha evolucionado el proyecto a lo largo del tiempo) y su valoración económica (la cuantificación del coste total de todas las horas invertidas en su desarrollo, así como el coste de la infraestructura con el que se ha llevado a cabo).

Aunque no se ha aplicado el estándar IEEE98 [39] para la elaboración de los planes de gestión de proyectos software, ya que no se pretende que este capítulo tenga la entidad ni la profanidad de un plan de gestión de este tipo, los apartados de este capítulo están fuertemente inspirados en el contenido de esta norma.

8.1 Planificación

A continuación, se muestra un detalle de las distintas tareas que se han llevado a cabo para la ejecución del proyecto junto con una pequeña descripción de ésta.

Posteriormente, se muestra el diagrama de Gantt del proyecto, para poder observar gráficamente los esfuerzos de dichas tareas a lo largo del proyecto.

1. Fase Inicio del proyecto

1.1. Definición de los objetivos del proyecto

En la fase inicial del proyecto se realizan una serie de entrevistas con el tutor del proyecto, con el fin de sacar en claro los objetivos del proyecto y realizar un plan inicial del proyecto para poder llevar un seguimiento continuo de éste.

1.2. Estudio del Arte

Una vez se ha definido el objetivo del proyecto, es necesario realizar unas sesiones de adquisición de información para refrescar aquellos temas relacionados con el proyecto y permitir obtener el conocimiento necesario para poder abordar el proyecto.

Esta fase es de vital relevancia ya que permitirá definir con mayor profundidad el objetivo del proyecto y poder definir las distintas líneas de trabajo a lanzar.

Debido a que desde un inicio se considera una fase vital para poder mantener el proyecto alineado con los objetivos, se ha estimado una duración de tres meses, que se consideran los necesarios para poder conocer los distintos temas que engloba el proyecto.

1.3. Preparación del Entorno de Trabajo

Una vez obtenida una base teórica del proyecto, es necesario preparar el entorno en el que poder comenzar a desarrollar el proyecto. Para ello, se estudiaron las distintas alternativas de implementación existentes con el fin de identificar las condiciones idóneas para el proyecto a abordar.

La fase de preparación del Entorno de Trabajo tiene una duración de un mes, ya que al tratarse de un sistema paralelo requiere de una infraestructura que garantizase las condiciones necesarias para el proyecto. Inicialmente, se preparó una plataforma auxiliar en donde poder iniciar las primeras fases del proyecto hasta contar con la infraestructura definitiva de implementación y pruebas.

Adicionalmente, dentro de esta fase, se idea el plan de experimentación que se va a lanzar para poder analizar el rendimiento del sistema paralelo.

1.4. Documentación

Fase que se ha lanzado en paralelo al resto. Se ha considerado que a la vez que se iba abordando una nueva tarea es imprescindible ir documentándola, con el fin de servir de guía posterior debido al gran alcance del proyecto y poder dejar reflejados todos los puntos relevantes.

2. Fase de Análisis del Algoritmo Secuencial

2.1. Estudio del Algoritmo Secuencial

Una vez preparado el Entorno de Trabajo, la primera tarea consiste en realizar un análisis exhaustivo del algoritmo secuencial del que se parte.

Debido a la naturaleza del proyecto, en donde se busca abordar un problema de paralelización, es vital conocer en detalle el algoritmo, por ello se ha invertido un importante tiempo en ello, aproximadamente tres meses. En estos tres meses es necesario en varias ocasiones realizar reuniones de trabajo con el tutor del proyecto, para poder comprender correctamente el funcionamiento de éste.

2.2. Identificación de los puntos de paralelización

Después de realizar un análisis exhaustivo del código del algoritmo secuencial, se detectan los posibles puntos de paralelización del algoritmo.

En la identificación de los puntos de paralelización, no sólo se basa en el análisis de cada una de las sentencias de código del algoritmo sino el análisis de herramientas de apoyo, tales como *gprof*, que permiten identificar aquellos fragmentos de código que siendo factibles de paralelización, permitirían obtener mayores beneficios al ejecutarse en paralelo.

2.3. Definición de la estrategia de paralelización

Una vez analizados los puntos de paralelización, analizando las necesidades de comunicaciones del problema y otros aspectos propios del algoritmo, se definen las distintas estrategias posibles de paralelización.

Una vez definidas las distintas estrategias de paralelización, se realiza un análisis de cada una de ellas con el fin de que se puedan contrastar y poder elegir la estrategia a implementar.

Debido a la gran variedad de posibilidades de estrategias a la hora de paralelizar el algoritmo, se han empleado aproximadamente 45 días, ya que se han analizado las ventajas y desventajas que podría tener cada una de estas estrategias.

2.4. Documentación

Al igual que en la Fase Inicial del proyecto, se inicia la documentación en paralelo a cada una de las tareas realizadas dentro del Análisis del Algoritmo Secuencial.

3. Fase de Paralelización MPI

3.1. Implementación reparto de carga de trabajo

Una de las primeras tareas dentro de la paralelización dentro de MPI, es la implementación del reparto de carga de trabajo. Esta fase es de gran importancia, ya que como se ha visto en otros apartados, el reparto de carga de trabajo es un factor de gran relevancia a la hora de implementar un sistema paralelo escalable.

3.2. Implementación fase de comunicación

La siguiente tarea que se aborda, es definir los requisitos de comunicación considerando la implementación de paso de mensajes. En esta fase, se definen qué datos son necesarios disponer en cada momento de ejecución del algoritmo y en función de la estrategia de paralelización considerada.

Se trata de una fase que se ha analizado profundamente, ya que en muchos sistemas paralelos, uno de los mayores hándicap a la hora de obtener mejores resultados tanto en tiempo de ejecución como en eficiencia es la necesidad de altos índices de comunicación, lo cual en el caso de MPI puede ser un factor crítico.

3.3. Construcción de tipos derivados

Una vez definida la fase de comunicación, es necesario definir los tipos derivados que permiten realizar envíos de datos con estructuras definidas en la implementación de éstos.

Se trata de una fase también muy importante, ya que se ha de buscar el equilibrio entre el tamaño del mensaje y el número de mensajes a enviar.

3.4. Pruebas de tipos derivados

La siguiente tarea, es realizar una serie de pruebas que garanticen que la implementación de los tipos derivados es correcta. Debido a que el testeo de las aplicaciones paralelas es más complejo que el testeo de aplicaciones secuenciales, se considera que es un punto lo suficientemente complejo e importante sobre el que realizar pruebas exhaustivas con el fin de buscar comportamiento anómalos.

3.5. Definición de variables públicas/privadas

Una vez definida la estrategia de paralelización y la estrategia de comunicación se realiza el análisis de las variables que permanecen privadas para los procesos y cuáles podrán ser públicas.

3.6. Implementación de la paralelización

Finalmente, se implementa el resto de la paralelización, obteniendo finalmente el algoritmo secuencial paralelizado con MPI.

3.7. Ejecución de pruebas

Antes de comenzar con la experimentación, es de vital importancia analizar los resultados del algoritmo paralelo para asegurar que el comportamiento de éste es el esperado. Para ello, se realiza una serie de pruebas que se comprueban con los resultados del algoritmo secuencial con el fin de poder identificar posibles casos anómalos.

3.8. Experimentación

La experimentación a realizar consiste en lanzar la aplicación paralela empleando un número distinto de procesadores, monitorizando en todo momento, los tiempos de ejecución obtenidos en cada una de las ejecuciones.

3.9. Documentación

Fase que tal y como se ha comentado anteriormente, se lanza en paralelo al resto de tareas.

4. Fase de Paralelización OpenMP

4.1. Implementación estrategia de planificación

Fase en el que en función de la estrategia tomada ya anteriormente en MPI, se implementa dicha estrategia esta vez haciendo uso de OpenMP.

4.2. Definición de variables públicas/privadas

Una vez definida la estrategia de paralelización y la estrategia de comunicación se realiza el análisis de las variables que permanecen privadas para los procesos y cuáles podrán ser públicas.

Fase de gran importancia puesto que en el caso de OpenMP, una definición incorrecta, supone un comportamiento del algoritmo indefinido.

4.3. Implementación paralelización

Finalmente, se implementa el resto de la paralelización, obteniendo finalmente el algoritmo secuencial paralelizado con OpenMP.

4.4. Ejecución de pruebas

Antes de comenzar con la experimentación, es de vital importancia analizar los resultados del algoritmo paralelo para asegurar que el comportamiento de éste es el esperado. Para ello, se realiza una serie de pruebas que se comprueban con los resultados del algoritmo secuencial con el fin de poder identificar posibles casos anómalos.

4.5. Experimentación

La experimentación a realizar consiste en lanzar la aplicación paralela empleando un número distinto de procesadores, monitorizando en todo momento, los tiempos de ejecución obtenidos en cada una de las ejecuciones.

4.6. Documentación

Fase que tal y como se ha comentado anteriormente, se lanza en paralelo al resto de tareas.

5. Fase de Comparativa MPI/OpenMP

5.1. Comparativa en Tiempos de Ejecución

La fase inicial en la comparativa de MPI y OpenMP es realizar un análisis de contraste en los tiempos de ejecución en ambas tecnologías. Con el fin de poder ir obteniendo posibles conclusiones acerca de la configuración más adecuada.

5.2. Comparativa de aceleración (speed-up)

A posteriori, se realiza una comparativa de los tiempos obtenidos en la experimentación esta vez considerando la aceleración.

5.3. Comparativa de eficiencia

Finalmente, se amplia la comparativa analizando el comportamiento de la eficiencia del sistema paralelo.

5.4. Conclusiones comparativa

Con las comparativas realizadas, se sacan las conclusiones correspondientes a bajo qué condiciones son las configuraciones más adecuadas del algoritmo.

6. Fase de Cierre del Proyecto

6.1. Definición de trabajos futuros

Fase en la que se detallan trabajos en los que se puede continuar investigando. Debido a que es necesario acotar el alcance del proyecto, esta fase pretende servir de guía de futuras líneas de trabajo a realizar.

6.2. Cierre documentación

Fase en la que una vez finalizado el trabajo, se revisa concienzudamente la documentación generado, con el fin de poder identificar posibles errores o identificar puntos en los que sea necesario realizar un análisis en mayor profundidad. Momento en el que además, se recogen los puntos más relevantes a reflejar dentro de la siguiente fase de la elaboración de la presentación para el Tribunal.

6.3. Preparación presentación (.ppt)

Fase en la que se debe realizar una presentación para su posterior exposición al Tribunal el día de la defensa del proyecto. Esta presentación servirá de guía para exponer los distintos análisis realizados dentro del proyecto y mostrar las conclusiones obtenidas con éste.

6.4. Presentación ante el Tribunal

Fase final del proyecto en la que se expone el proyecto al Tribunal encargado de evaluar el trabajo realizado.

A continuación, en la Figura 52 se muestra la estimación inicial realizada de cada una de las tareas del proyecto, en donde se puede observar el número de días dedicados a cada una de ellas.

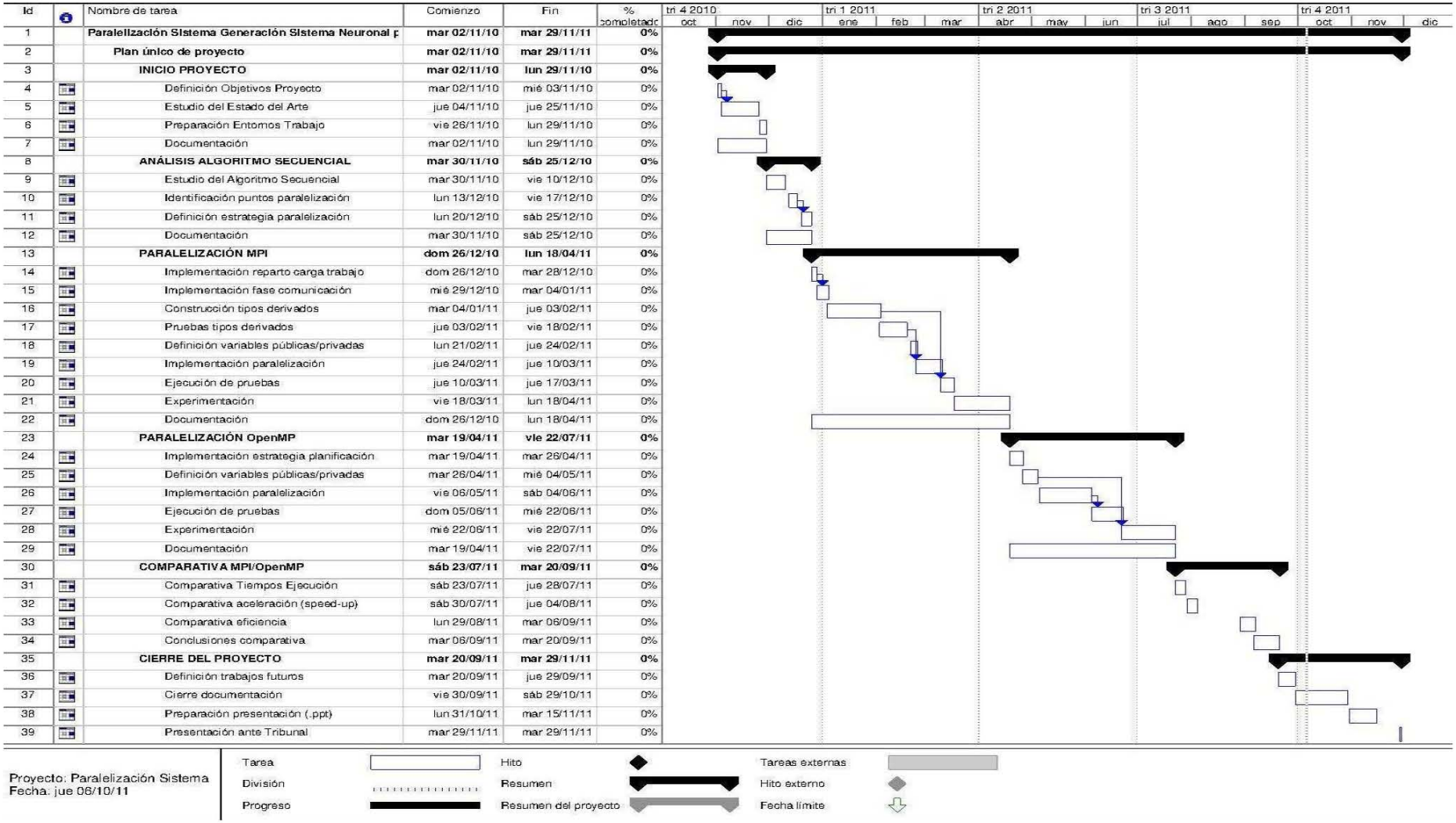

Figura 52 Planificación inicial del proyecto

A continuación, se muestra en la figura 53 la planificación del proyecto con los esfuerzos reales por cada una de las fases del proyecto.

Si se comparan ambas planificaciones se puede comprobar que la estimación inicial requería esfuerzos superiores a los que finalmente se han empleado, existe una desviación de 50 días.

Se puede observar que se ha invertido un esfuerzo mayor en las fases iniciales del proyecto de: Inicio del proyecto y Análisis del algoritmo secuencial. En la fase Inicio del proyecto, se ha dedicado un esfuerzo superior, 7 días, para la tarea de Estudio del Estado del Arte puesto que debido a la característica del algoritmo que había que paralelizar, ha sido necesario analizar conceptos teóricos que pudieran influir en la posterior paralelización del sistema. En la del Análisis del algoritmo secuencial, 17 días, se ha debido a que se consideró una tarea crítica para poder abordar correctamente el resto del proyecto, por lo que, fueron necesarias distintas sesiones de trabajo junto con el tutor del proyecto para poder comprender en profundidad el funcionamiento de éste.

Existe una reducción importante en la fase de Experimentación tanto para MPI como para OpenMP, 17 días y 16 días respectivamente. El motivo de esta reducción es que la planificación que se realizó inicialmente del esfuerzo de la experimentación no se realizó considerando la reducción en el tiempo de ejecución que se ha obtenido con la paralelización, lo cual provoca que las pruebas a realizar tuvieran una duración mucho menor, recordar que en el caso de MPI se llegó a obtener una mejora de un 72% del tiempo y en el caso de OpenMP una mejora de un 64% del tiempo.

En el caso de la fase de Cierre del Proyecto, existe un esfuerzo superior al inicialmente planificado, 27 días, debido a que existieron distintas reuniones de trabajo con el tutor para poder cerrar ciertos puntos tanto de la memoria como de la presentación que posteriormente se realizó para la presentación del proyecto ante el Tribunal.

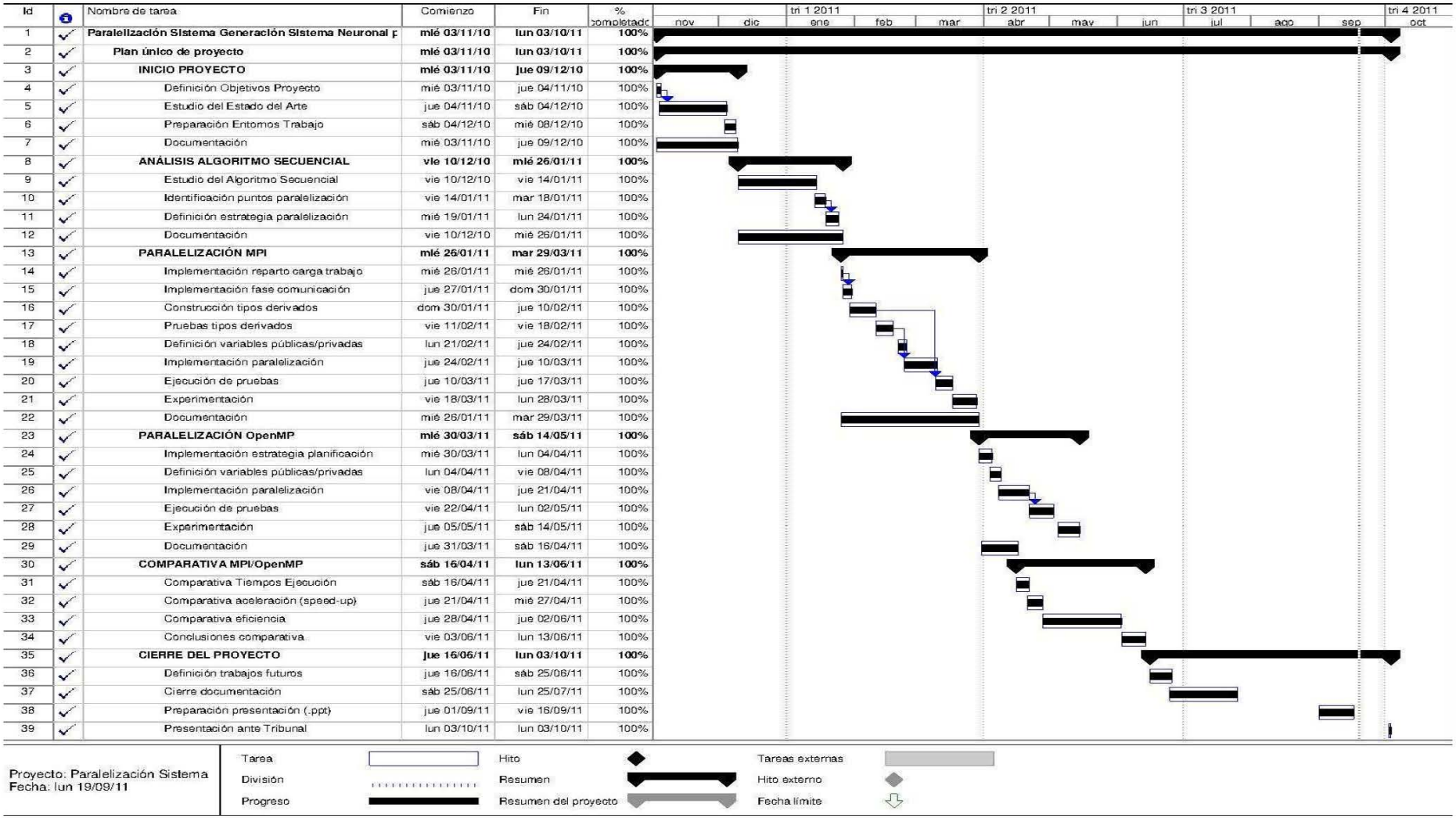

Figura 53 Planificación final del proyecto

8.2 Valoración Económica

Uno de los primeros costes a considerar dentro del proyecto es toda la arquitectura necesaria para poder realizar la experimentación a realizar, **coste hardware**. Para poder lanzar la experimentación, ha sido necesario contar con un equipo multiprocesador HP Proliant ML350 con cuatro procesadores dual-core. El precio de dicho equipo es de: **3.190€ + IVA**.

Adicionalmente al coste de hardware comentado, es necesario analizar el coste de las posibles **licencias de software** necesarias para poder realizar el desarrollo del proyecto. El proyecto se ha implementado empleando MPI y OpenMP que resultan gratuitos. Además, para la implementación se han empleado programadas de código libre, lo cual permiten reducir los costes de licencias dentro del proyecto.

Finalmente, es necesario analizar las horas empleadas en cada una de las fases, y los distintos perfiles participantes en el proyecto con el fin de obtener el **coste de personal**. Dicho coste se resumen a continuación:

Tareas	Horas	€/h	Total
Analista	6.849,00	30,00	205.470,00
Director proyecto	684,90	60,00	41.094,00
Total	**7.533,90**	**32,73**	**246.564,00**

Tabla 34 - Detalle coste de personal del proyecto

Por lo que, el coste total del proyecto es de:

Naturaleza	Coste (€)
Coste HW	3.764,20
Coste SW	0,00
Coste Personal	246.564,00
Total	**250.328,20**

Tabla 35 - Coste total del proyecto

IVA del 18% incluido.

Apéndice I: Manual de Instalación de MPI

A continuación se detallan los requisitos y pasos necesarios para realizar una instalación satisfactoria de MPICH2 en los sistemas operativos de Linux y Windows.

Instalación en Linux:

Requisitos

Compilador de C/C++

Instalación

1. Descargar el código fuente de http://www-unix.mcs.anl.gov/mpi/mpich

$ wget http://www-unix.mcs.anl.gov/mpi/mpich/downloads/

2. Desempaquetar el fichero .tar.gz

Se recomienda descomprimir el fichero en $HOME/libraries, de tal forma que dentro de ese directorio quede el subdirectorio mpich2, teniendo la siguiente estructura:
$HOME/libraries/mpich2

$ tar xvfz mpich2

3. Elegir un directorio de instalación. (Por defecto es /usr/local/bin)

$ mkdir $HOME/mpich2-install

4. Elegir un directorio temporal en el que compilar los fuentes

$ mkdir /tmp/usr/
$ mkdir /tmp/usr/mpich2

5. Preparación del entorno de compilación de los fuentes, instalando en el sistema los paquetes que son necesarios.

$ su
$ aptitude install gcc g++ make open-ssl libssl-dev

Dónde, gcc y g++ son los compiladores de C y C++ respectivamente y open-ssl y libssl-dev son los paquetes necesarios para dar soporte MD5 ya que es requerido por el manejador de procesos smpd.

6. Configurar MPICH2

$ cd /tmp/usr/mpich2-1.0.5p4

$ $HOME/libraries/mpich2-1.0.5p4/configure -with-pm=smpd -prefix=/home/usr/mpich2-install | tee configure.log

Nótese que se va a configurar MPICH2 utilizando como manejador de procesos smpd. En el siguiente apartado se analiza qué es y qué otras alternativas posibles existen.

7. Construir MPICH2

$ make | tee make.log

8. Instalar los comandos MPICH2

$ make install | tee install.log

9. Añadir el directorio de binarios al path
    a. Comando export

$ PATH=$PATH:/home/usr/mpich2-install/bin
$ export PATH

    b. Actualizar el fichero $HOME/.bashrc

#bin mpi path
PATH=/home/usr/mpich2-install/bin:"${PATH}"
#C include mpi path
PATH=/home/usr/mpich2-install/include:"${PATH}"

10. Añadir la frase de paso de acceso en $HOME/.smpd

$ vi $HOME/.smpd

Incluir la siguiente línea:

phrase=be happy

Es conveniente que el fichero sólo pueda ser leído y modificado por el propio usuario. Para ello, hay que modificar sus permisos:

chmod 700 $HOME/.smpd
-rwx--- 1 usr usr 15 2007-07-05 21:31 .smpd

Instalación en Windows:

Requisitos

1) MS Development Enviroment
2) Visual Studio o gcc para compilar los programas MPI en C/C++
3) Permisos de administrador, para que se pueda instalar el proceso smpd.exe en las máquinas y se permita la copia de las librerías en el directorio de Windows.

Instalación

Distribución binaria

1) Descargar e instalar en todas las máquinas la distribución MPICH2 de 32 bits o 64bits según la arquitectura de su computador. El instalador al finalizar copiará las librerías necesarias dentro del directorio windows\system32, además creará la estructura de directorios en la máquina:

    - path_instalación\bin
    - path_instalación\include
    - path_instalación\lib

2) Tras finalizar el proceso de instalación, editar la variable “path” de las variables de entorno del sistema, para añadir los ejecutables de la distribución que se encuentran en el directorio “bin” de la instalación.

3) Si se tiene instalado un firewall, es necesario añadir las excepciones para los programas mpiexec.exe y smpd.exe.

Apéndice II: Manual de Instalación de OpenMP

Instalación en Linux:

GCC dispone de OpenMP desde la versión 4.2, simplemente con disponer instalada una versión de gcc igual o posterior a ésta será posible compilar y ejecutar programas con OpenMP añadiendo el flag –fopenmp al compilador gcc.

Instalación en Windows:

- **Visual Studio 2008 y Visual Studio 2010**

La implementación de OpenMP 2.0 es soportado por Visual Studio 2008 y 2010. Para poder activar la compilación y ejecución de OpenMP en Visual Studio es necesario ir a las Propiedades del proyecto “Propiedades de configuración” -> “C/C++”->”Lenguaje” y marcar “Sí (/openmp)” en la casilla de “Compatibilidad de OpenMP”.

Nota: Existe la posibilidad de activar la compilación y ejecución de OpenMP en versiones anteriores al Visual Studio 2008.

- **Eclipse**

El entorno de desarrollo de eclipse no cuenta con soporte para OpenMP por defecto. Pero es posible habitarlo siguiendo los siguientes pasos:

1. Instalar PTP (*Parallels Tools Plataform*) de Eclipse a través de *Help, Install new Software*.

2. Reiniciar el entorno eclipse para que disponga de esta nueva plataforma instalada

3. Dentro de las propiedades del proyecto en cuestión, *“Propiedades del Proyecto” > Tool Chain Editor -> Current Toolchain = Linux GCC*

4. Añadir la línea de comandos para el soporte de OpenMP:

Propiedades del proyecto -> Settings -> Tool Settings -> GCC C++ Compiler -> Miscellaneous

5. Añadir *“-fopenmp”* en *Other Flags*

Apéndice III: Manual de Usuario

Manual de usuario MPI

Para la ejecución del algoritmo empleando MPI es necesario disponer de una estructura de carpetas:

Carpeta raíz
Código Fuente (.c, *.h)*
Carpeta Pruebas
Fichero Serie Temporal
Fichero script
Fichero configuración
Fichero mynodes
Carpeta "nets"

Abrir el terminal y situarse dentro de la carpeta raíz anteriormente mencionada y ejecutar la siguiente línea:

time mpirun –machinefile mynodes –np X ag –P FicheroConfiguración –t FicheroSerieTemporal –n 0

donde:

- *mynodes*: se trata del fichero donde se especifica los equipos donde se desea ejecutar el algoritmo
- *X*: se trata del número de procesadores que se quiere lanzar en la paralelización
- *FicheroConfiguración*: se trata del fichero en donde se encuentra la configuración del Sistema
- *FicheroSerieTemporal*: se trata del fichero donde se encuentran los valores de la serie temporal

Manual de usuario OpenMP

Para la ejecución del algoritmo empleando MPI es necesario disponer de una estructura de carpetas:

Carpeta raíz
Código Fuente (.c, *.h)*
Carpeta Pruebas
Fichero Serie Temporal
Fichero script
Fichero configuración

Fichero mynodes
Carpeta “nets”

Abrir el terminal y situarse dentro de la carpeta raíz anteriormente mencionada y ejecutar la siguiente línea:

time ./ ag –P FicheroConfiguración –t FicheroSerieTemporal –n 0 –np X –pl Planificación –c Y

Donde:

- *FicheroConfiguración*: se trata del fichero en donde se encuentra la configuración del Sistema
- *FicheroSerieTemporal*: se trata del fichero donde se encuentran los valores de la serie temporal
- *X*: se trata del número de procesadores que se quiere lanzar en la paralelización
- Planificación: se trata de la estrategia de planificación a incluir en la directiva *schedule*
- *Y:* se trata del tamaño del *chunk* a emplear en la directiva schedule

Bibliografía

[1] John Haugeland - *Artificial Intelligence: The Very Idea* (1985). Cambridge, Mass.: MIT Press.

[2] Eugene Charniak and Drew McDermott - *Introduction to Artificial Intelligence*. Addison-Wesley, 1987.

[3] Richard E. Bellman - *An Introduction to Artificial Intelligence: Can Computers Think?*. Boyd & Fraser Publishing Company, 1978.

[4] Patrick Henry Winston - *Artificial Intelligence, Third Edition*.

[5] Kevin Knight, Elaine Rich, B. Nair - *Artificial Intelligence*. McGraw-Hill 1991

[6] Nils J. Nilsson. *Artificial Intelligence: A new Síntesis*. San Francisco: Morgan Kaufmann, 1998

[7] George Box, Gwilym M. Jenkins, and Gregory Reinsel. *Time Series Analysis: Forecasting & Control (3rd Edition)*. Prentice Hall, 3rd edition, February 1994.

[8] Daniel Peña. *Análisis de series temporales*. Alianza Editorial, Madrid, 2010.

[9] Gallistel, C. R., Brown, A., Carey, S., Gelman, R., and Keil, F. (1991) *Lessons from animal learning. In R. Gelman & S. Carey (Eds.) The epigenesis of mind. Hillsdale, NJ: Erlbaum. pp. 3-37.*

[10] John M. Pearce. *"An Introduction to Animal Cognition"*. Psychology Press.

[11] E.Plaza. “Tendencias en Inteligencia Artificial: hacia la cuarta década”. In A. del Moral, editor, Nuevas tendencias en Inteligencia Artificial, páginas 379-425. U.Deusto, 1992.

[12] Holland, John H. (1975). *Adaptation in Natural and Artificial Systems (Cambridge (Mass)*: The MIT Press, 1992. [Edición por *MIT Press* de la edición original de *The University of Michigan*])

[13] Ramesh Sharda. *Neural Networks for the ms/or analista: An application bibliography*. INTERFACES, 24(2): 116.130, 1994.

[14] Enrique Alba. *Análisis y Diseño de Algoritmos Genéticos Paralelos Distribuidos.PhD thesis*, Universidad de Málaga, 1999.

[15] Erick Cant´u-Paz. *A survey of parallel genetic algorithms. Calculateurs Paralleles, Reseaux et Systems Repartis*, 10(2):141–171, 1998.

[16] Aleksander, Igor. *Neural Computing Architectures*. MIT Press, Cambridge, MA, 1989

[17] Diederich, J. ed. *Artificial Neural Networks: Concept Learning*. Computer Society Press, Los Alamitos, CA, 1990

[18] Xabier Basogain Olabe. *Redes Neuronales Artificiales y sus Aplicaciones*. Publicaciones de la Escuela de Ingenieros, 1998

[19] Pedro Isasi Viñuela , Inés M. Galván León. *Redes de Neuronas Artificiales. Un enfoque Práctico*. Editorial: Pearson Prentice Hall

[20] Flynn, M. (1972). *"Some Computer Organizations and Their Effectiveness"*. IEEE Trans. Comput. C-21: 948

[21] Duncan, Ralph (1990). *"A Survey of Parallel Computer Architectures"*. IEEE Computer: 5–16.

[22] Kai Hwang. *Advanced computer architecture: Parallelism, scalability, programmability.* McGraw-Hill Higher Education ©1992

[23] Mehdi R. Zargham. *Computer Architecture, single and parallel systems.* Prentice-Hall, Inc. Upper Saddle River, NJ, USA ©1996

[24] Herbert Hoeger. Introducción a la Computación Paralela. Centro Nacional de Cálculo Científico Universidad de Los Andes - CeCalCULA - Mérida – Venezuela

[25] William Gropp. *Tutorial on MPI: The Message-Passing Interface*

[26] David Walker. *MPI: from Fundamentals to Applications*

[27] Message Passing Interface Forum. MPI: A Message-Passing Interface standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8, 1994

[28] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Bob Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine---A User's Guide and Tutorial for Network Parallel Computing*. MIT Press, 1994

[29] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. Mc Graw-Hill. Capítulo 17.

[30] Rohit Chandra, Dave Kohr, Leonardo Dagum, Ramesh Menon, Dror Maydan, Jeff McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann

[31] B. Chapman, G. Jost, R. van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, 2008

[32] Ruby Lee. *Precision Architecture. IEEE Computer. Vol. 22 No. 1, pp. 78-91*. Jan. 1989

[33] Wilkinson, Allen. *Parallel Programming*. Prentice Hall 2005.

[34] A. Grama, G. Karypis, V. Kumar, and A. Gupta, *Introduction to Parallel Computing*, Addison Wesley, second ed., 2003

[35] John L. Gustafson. *Reevaluating Amdahl's Law*. Communications of the ACM 31(5), 1988. pp. 532-533

[36] Xian-He Sun, Nabil Kamel, Lionel M. Ni. *SIGMOD '89 Proceedings of the 1989 ACM SIGMOD international conference on Management of data.* ACM New York, NY, USA ©1989

[37] Gene Amdahl, *"Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities"*, AFIPS Conference Proceedings, (30), pp. 483-485, 1967.

[38] Smith, J.E., Pleszkun, A.R.. *Implementing precise interrupts in pipelined processors*. Wisconsin Univ., Madison, WI

[39] Institute of Electrical and Electronics Engineers: "*IEEE Standard for Software Project Management Plans*". IEEE, 1998. ISBN: 0738114472.

[40] Antonio Moreno Ribas, Eva Armengo Voltas y Javier Béjar: "Aprendizaje Automático". Universidad Politécnica de Cataluña.

[41] Quinn, Michael J. "*Chapter 2: Parallel Architectures*." *Parallel Programming in C with MPI and OpenMP.Boston*: McGraw Hill, 2004. ISBN 0-07-282256-2

[42] Amir D Aczel. *God´s equation: Einstein, relativity, and the expanding universe*. MFJ Books, New York, 1999.

[43] Markos Papageorgiou, Apostolos Kotsialos and Antonios Poulimenos. *Long-term sales forecasting using holt-winters and neural network methods*. Journal of Forecasting.

[44] S. Makridakis, S.C. Wheelwright, and R.J. Hyndman. *Forecasting methods and applications*. John Wiley & Sons, USA, 3rd edition, 2008.

[45] Ian Nunn and Tony White. *The application of antigenic search techniques to time series forescasting*. In Proceedings of the 2005 conference on Genetic and evolutionary computation, GECCO 05, pages 353-360, New York, USA, 2005. ACM.

[46] Paulo Cortez. *Data mining with neural networks and support vector machines using the r/rminer tool*. In Proceedings of the 10th industrial conference on Advances in data mining: applications and theoretical aspects, ICDM, Berlin, Heidelberg, 2010. Springer-Verlag.

[47] Sven F. Crone and Nikolaos Kourentzes. *Feature selection for time series prediction – a comnied filter and wrapper approach for neural networks*. Neurocomput, June 2010.

[48] José Luis Aznarte M., José Manuel Benítez, and Juan Luis Castro. *Smooth transition autoregressive models and fuzzy rule-based systems: Functional equivalence and consequences*. Fuzzy Sets and Systems, 2007.

[49] M. Stepnicka, J. Peralta, P. Cortez, Lenka Vavrickova and G. Gutierrez. *Forecasting seasonal time series with computational intelligence: contribution of a combination of distinct methods*. In EUSFLAT 2011, 2011.

[50] N.K. Kasabov and Qun Song. *Denfins: dynamics envolving neural-fuzzy interference system and its application for time-series prediction*. Fuzzy Systems, IEEE Transactions, April 2002.

[51] Juan Peralta, German Gutierrez and Araceli Sanchis. Adann: automatic design of artificial neural networks. In Proceedings of the 2008 GECCO conference companion on Genetic and evolutionary computation, GECCO, New-York, USA, 2008. ACM.

[52] http://www.neural-forecasting competition.com. *Forecasting competition for neural networks & computational intelligence*. Último acceso en Marzo del 2011.